VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Windows Phone a OpenStreetMaps

Windows Phone and OpenStreetMaps

2013 Martin Škuta

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Martin Škuta**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Windows Phone a OpenStreetMaps
Windows Phone and OpenStreetMaps

Zásady pro vypracování:

V rámci této diplomové práce se bude student zabývat problematikou vybranými částmi frameworku připravovaného na VŠB-TU Ostrava pro vyhledávání nejvhodnějších trasy pro navigaci (routování) dopravního vozidla. Student bude řešit ve vyvíjeném frameworku dvě oblasti. Jednou je předzpracování a aktualizace navigačních dat jakými jsou OpenStreetMap. V druhé oblasti se bude zabývat vývojem aplikace pro mobilní platformu Windows Phone.

Jednotlivé body zadání:
1. Prostudování formát navigačních dat OpenStreeMap.
2. Implementace aplikaci pro nahrávání OpenStreetMap do prostorové databáze s možností aktualizace dat.
3. Implementace knihovny pracující s prostorovou databází s uloženými navigačními daty, která bude poskytovat potřebné rozhraní pro další moduly vytvářeného frameworku.
4. Vytvoření Windows Phone aplikace pro routování dopravního vozidla.
5. Experimenty zaměřené na čas zpracování dat.

Seznam doporučené odborné literatury:

[1] OpenStreetMap, otevřená wiki-mapa světa: http://www.openstreetmap.org/
[2] Windows Phone| Dev Center: http://dev.windowsphone.com/en-us

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Jan Martinovič, Ph.D.**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2013

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 19. července 2013 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 19. července 2013 .....

Rád bych na tomto místě poděkoval všem mým spolupracovníkům z projektu Floreon, především panu doktorovi Janu Martinovičovi.

Abstrakt

Tato práce je součástí rozpracovaného frameworku připravovaného na VŠB-TUO, který má umožnit navigaci dopravních vozidel. První část této práce se věnuje tvorbě nástroje na zpracovaní dat z projektu OpenStreetMaps a vytvořením tak grafu silnic pro Evropu, minimalizaci daného grafu a také vytvoření bodů zájmů. Druhá část se pak zabývá tvorbou Windows Phone aplikace, která s danými daty pracuje.

Klíčová slova: PostGIS, Windows Phone, OpenStreetMaps, framework, geodata

Abstract

This thesis is part of a framework created at VŠB-Technical University of Ostrava, which will be used for navigation of vehicles. Thesis is divided into two parts, the first part focuses on developing of tool for extraction and indexing of geo datas from OpenStreetMaps project and creation of routable graph of highways in Europe, minimalization of that graph and also creation of database of points of interests. Second part focuses on developing of Windows Phone app, that will use thoose datas.

Keywords: PostGIS, Windows Phone, OpenStreetMaps, framework, geodata

Seznam použitých zkratek a symbolů

GPS	–	Global positioning system
OSM	–	Open street map
UML	–	Unified Modeling Language. Jazyk pro vizualizaci, specifikaci a navrhování programových systémů.
DB	–	Databáze
SQL	–	Structured Query Language. Jazyk pro dotazování dat z databáze.
DDL	–	Data Definition Language. Jazyk používaný v databázích, kterým se definují např. tabulky. Typicky je to třeba příkaz CREATE.
POI	–	Point Of Interest. Anglicky bod zájmu, zkratka přejímaná i do jiných jazyků.
POCO	–	Plain Old CLR object. Klasické objekty jak je známe např. z jazyka C#
CLR	–	Common Language Runtine. Je virtuální stroj, kterým se překládají a spouští programy napsané v jazycích založených na .NET Frameworku.

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Digitalizované mapy již dnes patří k samozřejmosti, ať už to jsou navigační systémy pro raketové systémy, letadla, vozidla či chodce, zobrazování adres, nerostných bohatství, katastrální mapy apod., tedy digitální mapy může dnes využít úplně každý a taky se tak děje díky inovacím v technologiích a zveřejňování mapových dat. Je to hlavně díky americké armádě, která se v roce 1983 [9], potom co bylo sestřeleno korejské civilní letadlo, které se ztratilo na sovětském území, rozhodla zpřístupnit svůj družicový polohový systém GPS a způsobila tím, že v dnešní době existují kolem nás miliony zařízení, která dokáží zaměřit svoji polohu vůči zeměkouli, tzv. "Location-aware devicies" [11]. Díky zpřístupnění GPS a inovacím ve vývoji výroby čipů přijímající GPS signál, jsme dnes schopni velmi jednoduše mapovat povrch a ukládat jej v digitální podobě do nejrůznějších databází. Čip přijímající GPS signál je dnes, tak malý, že jej najdeme dnes opravdu všude, hlavně tedy v našich kapsách v mobilních telefonech, což nám otevírá neomezené možnosti využití v aplikacích, které tak můžou v kontextu s polohou nabízet mnohem přesnější a relevantnější obsah a v neposlední řadě nám pomoct při navigaci v okolí. Díky tomu dnes kromě již zmíněných mobilních aplikací, které využívají polohu, máme také velmi kvalitní mapové podklady, protože cena těchto zařízení je malá, přesnost je již díky inovaci výrobního procesu dostačující a tak se víceméně každému otevírá možnost digitálně mapovat vybrané území a zájmové body a firmy jako Google, Nokia, či Microsoft se toho snaží co nejvíce využívat.

To, že každý může mapovat okolí, je na jednu stranu dobrá zpráva, ale na druhou stranu to s sebou nese nově vzniklé problémy. Hlavní problém je to, že když každý může mapovat zemský povrch, kam bude tyto data ukládat a jak se k nim dostane někdo jiný, kdo by je chtěl taky využít? To bylo ještě do nedávna celkem složité a do jisté míry to je stále problém. Například v dnešní době velmi oblíbená geolokační hra Geocaching, kde uživatelé schovávají tzv. "kešky", uloží do mapy jejich polohu a další uživatelé se ji snaží najít a nechat tam třeba vzkaz, či nějakou drobnost pro další hráče, kteří ji také hledají. Tady se problém projevuje nejdříve z pohledu hráče, který vytvoří novou "kešku", protože ten samozřejmě chce, aby ji našlo co nejvíce dalších hráčů. V nejčastějším případě polohu nové "kešky"uloží na nejnavštěvovanější portál o geocachingu. A to samé platí obráceně, když hráč chce jenom hledat "kešky". Kam se má podívat, aby jich mohl najít co nejvíce, neboli, na kterém serveru je uloženo nejvíce poloh "kešek". V obou případech by bylo ideální, kdyby se všechny "kešky"ukládaly na jeden jediný server a všechny geolokační portály z něj čerpaly a úplně nejlepší by bylo, kdyby ten server tyto data poskytoval zadarmo. Přesně tenhle typ problému se snaží řešit OpenStreetMap projekt, který se od začátku snaží kolaborací uživatelů vytvořit největší editovatelnou mapu světa, přístupnou pro kohokoliv a to vše zadarmo. A hlavně proto, že tato data jsou volně k dispozici bez jakýchkoli poplatků i pro komerční využití stává se tento projekt velmi důležitý co se mapování týče a dnes můžeme najít spoustu služeb a frameworků postavených právě nad daty z tohoto projektu. Kvůli tomu, že v OpenStreetMap projektu může uložit data úplně každý roste tato databáze velmi rychle a je zde díky tomu opravdu mnoho zajímavých dat. Do tohoto projektu přispívají i velké společnosti jako Automotive Navigation Data,

která dodala celou silniční síť Nizozemska [6], Yahoo dodalo satelitní snímky [5] a kvůli cenám a poplatkům jiných mapových dat, jakou jsou například Google Maps, se hodně i velkých společnosti a korporací rozhodlo přejít právě na data z OSM [8]. Například to byl Foursquare, Apple ve svých mapách pro mobilní operační systém iOS6 částečně vužíval OSM data [13] nebo z těch velkých třeba ještě Craiglist [7]. Právě proto, že data z OSM už se zdají být použitelné, rozhodli jsme se pro náš projekt, potažmo diplomovou práci vybrat jako zdroj právě data z OSM. Z těchto dat potom vytvářím graf silnic, po kterém půjde navigovat vozidlo, získat databázi zájmových bodů jako restaurace, divadla apod. a taky veškeré dostupné poštovní adresy. Dále v rámci této diplomové práci budou tyto data využita pro výpočet dojezdové doby a dostupnosti záchranářských či hasičských vozidel a nakonec vše využito v aplikaci pro chytré telefony běžící s operačním systémem Windows Phone.

2 Projekt OpenStreetMap

Jako zdroj dat pro připravovaný framework jsme tedy vybrali geodatabázi z projektu OpenStreetMap, proto bych chtěl ještě popsat tento projekt trochu blíže, aby bylo jasné proč přesně jsme tento zdroj dat vybrali a taky aby kapitola o zpracování dat dávala smysl.
OpenStreetMap projekt založil v roce 2004 Steve Coast z Velké Británie, ale projekt začal sbírat data až v roce 2006, kdy jej začala podporovat stejnojmenná organizace, která začala podporovat tvorbu dat a snažila se nasbírané data šířit dál a zpřístupňovat komukoliv. Hlavní myšlenkou je zmapovat celý svět, neboli posbírat co nejvíce dat se zaznamenanou polohou a tyto nasbírané data zpřístupnit všem a to celé zadarmo. Jedná se tedy o wikipedii geodat, kde kdokoliv může vytvořit nová data nebo změnit, již existující. Hlavním motivem pro výběr dat z OpenStreetMap je tedy to, že tyto data jsou úplně zadarmo a mají v sobě data o celém světě a to ne jen konkrétních geodat jako jsou například silnice, ale všeho. V OpenStreetMap databázi tak najdeme data téměř pro jakýkoliv projekt zabývající se geodaty. Najdeme tam třeba data o řekách, nerostných surovinách, všech možných bodů zájmů, či již zmiňovaných "kešek"pro geocaching, polohy konaných konferencí atp. V podstatě v OpenStreetMap databázi můžeme najít úplně všechno co se dá nějak spojit s polohou na zemi. Právě z tohoto důvodu jsme se rozhodli použít data z OpenStreetMap, protože jakmile se vytvoří způsob jak číst určitá data z OSM, je pak už snadné načíst další data v případě rozšíření projektu o jiná geodata. Tyto výhody s sebou ovšem nesou i nevýhody. Kvůli tomu, že do této databáze může přispívat opravdu kdokoliv, je velmi těžké ověřovat pravost dat a asi největším problém OpenStreetMap je redundance a konzistence. Nejprve však o formátu dat v databázi OSM.

2.1 Formát dat v OSM

V OSM databázi najdeme 3 základní elementy (Data primitives), na kterých je postavená celá databáze. Jsou to: Uzel, Cesta a Relace.

- **Uzel**, neboli Node, je bod na zemském povrchu, který má souřadnice ve formátu WSG84. Je to základ všech dalších elementů a od poslední verze je možné volitelně ukládat i nadmořskou výšku.
- **Cesta**, neboli Way, je lineární element, kterým se tvoří linie nebo polygony. Ve své podstatě je to setříděný seznam elementů typu uzel. Od linie a polygonu se rozlišuje tak, že pokud se první a poslední uzel v seznamu rovnají, jedná se o uzavřenou linii, neboli polygon. Existuje zde omezení, že seznam uzlů v cestě může mít minimálně 2 uzly a maximálně 2000 uzlů. Pokud je třeba vytvořit linii, či polygon s více než 2000 uzly použije se relace.
- **Relace**, neboli Relation, umožňuje tvořit vazby a relace mezi ostatními elementy. Ve své podstatě je to opět setříděný seznam členů, kterými můžou být uzly, cesty a nebo samotné relace. Relace se nejčastěji používají k vytvoření tématické vazby

mezi objekty, například Velká kanadská jezera jsou dobrý příklad, u této relace by byly členové jednotlivé polygony jezer. Nebo se také používají k vytvoření velkých polygonů, například polygony států, kde jednotlivé linie hranice státu jsou členy relace a polygon se opět uzavře jako v případě cest, tehdy, když se první a poslední člen seznamu rovnají.

Dále existují v databázi objekty typu User, zaznamenané trasy GPX, Friends a další. Tyto objekty nejsou pro tuto práci podstatné, jsou to informace o uživatelích, jejich uložených trasách, přátel apod. Tyto data nejsou součástí exportů z databáze a většina těchto dat je přístupných pouze přes web nebo webové API a většinou jsou přístupné informace pouze o přihlášeném uživateli nebo uživateli od kterého máme povolení data číst.

2.2 Společné vlastnosti

Třemi základními elementy jsou definovány všechny geoobjekty v databázi OSM. To by ovšem pro uložení například silnice nestačilo, protože chybí například způsob, jak uložit jméno cesty, ulice apod., dále chybí u těchto tří elementů verzování, jelikož data může upravovat a přidávat každý, musí existovat nějaký způsob, jak například vrátit změnu pokud někdo něco uložil chybně nebo pokud se například cesta změnila a postavil se na ni například kruhový objezd místo křižovatky. Proto ještě v OSM existují tzv. společné vlastnosti neboli anglicky common atributes. Tyto atributy jsou společné pro všechny tří zmíněné základní elementy.

První a nejdůležitější obecný atribut, který mají všechny elementy společný je id, které daný element jednoznačně identifikuje viz. definice 2.1.

Definice 2.1 *Id je jednoznačný identifikátor objektu v databázi OSM a má tyto vlastnosti:*

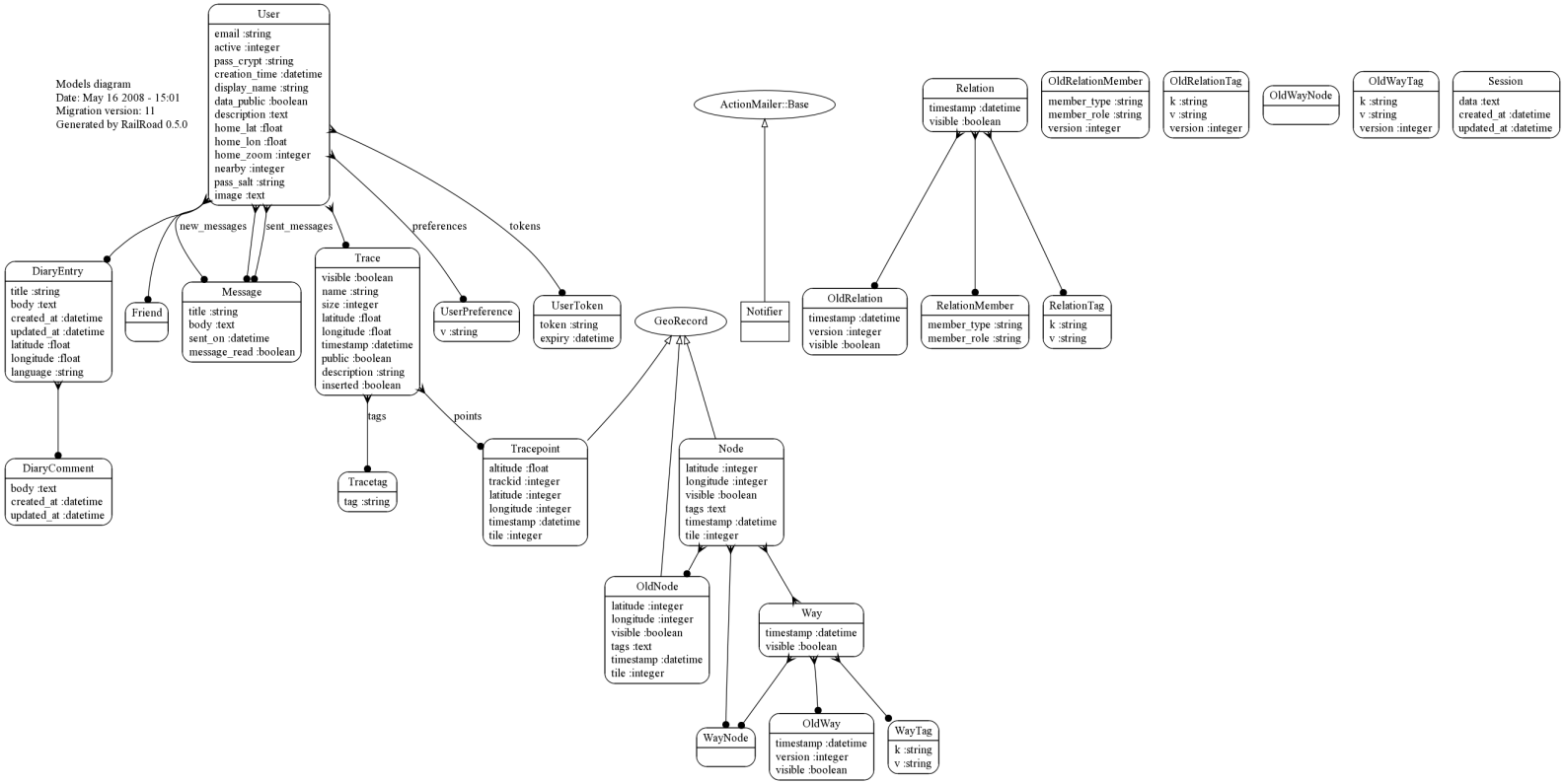
- *je to znaménkový 64bitový integer*
- *každý objekt má svůj vlastní prostor pro atribut Id. Proto v databázi například určitě existuje uzel a cesta s Id 100, ale to neznamená, že mají něco společného*
- *hodnota 0 se používá pro nové objekty, které ještě v databázi neexistují*
- *záporné hodnoty jsou u objektů v databázi zakázané, používají se pouze k číslování nově vytvořených objektů v souborech pro aktualizaci databáze, tzv. diffs, které používají OsmChange formát.*

V projektu OSM se udržují všechny data od počátků projektu, proto existuje historie a musí existovat i verzování jednotlivých elementů. Pro verzování a historii existuje v databázi sada těchto společných atributů:

- user - zobrazované jméno uživatele, který přidal, či naposledy upravil daný objekt, v době změny (jméno si může uživatel změnit, proto se ukládá)

- uid - jednoznačné id uživatele
- timestamp - počet milisekund od 1.ledna 1970 do vytvoření daného elementu
- version - verze objektu, první verze je vždy 1
- visible - boolean označující zda je element smazaný. Pokud je nastavený na false, tak to znamená, že objekt byl smazán, takový objekt najdeme pouze v historii do exportu dat z OSM se dávají pouze elementy s visible nastaveným na true.
- changeset - identifikátor changesetu. Changeset je kolekce všech elementů, které byly upraveny, či přidány. Například pokud chci přidat na cestu zmiňovaný kruhový objezd, musím smazat body tvořící křižovatku a přidat body tvořící kruhový objezd. Všem těmto změnám přiřadím jedno Id changesetu a changeset označím jako přidání kruhového objezdu.

Díky těmto atributům je zaručena identifikace objektů a jejich historie, avšak nejdůležitější z pohledu tvůrců aplikací postavených na datech z OSM jsou tzv. štítky, neboli tags. Ke každému ze tří základních elementů, lze připojit neomezený počet štítků. Štítek je jednoduchá dvojce klíče a k němu přiřazená hodnota, například štítek jméno, popis a podobně. Díky těmto štítkům můžeme objektům přidávat meta informace a odlišit tak o jaký element se vlastně jedná, jaký objekt těmito daty popisuje. Např. jestli ta linie je cesta, hranice pozemku, či stěna domu a podobně.
Všechny tyto elementy a atributy jsou uložené v relační databázi, která je založená na databázovém enginu PostgreSQL. Zjednodušené schéma celé databáze si můžete prohlédnout na Obrázku 1



Obrázek 1: Zjednodušené schéma databáze OSM. Zdroj: http://wiki.openstreetmap.org/

3 Import a zpracování OSM dat

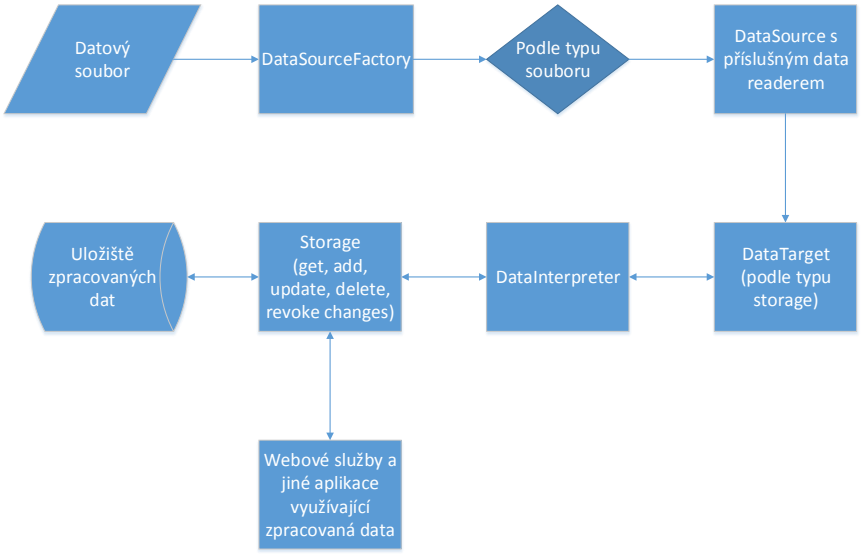
Jak jsem již říkal, OSM zpřístupňuje svá data pro všechny. Existuje několik různých způsobů a formátů, jak data získat. Prvním z nich je tzv. planet.osm, což je jeden velký soubor se všemi uzly, cestami a relacemi v jejich poslední verzi, který se aktualizuje jednou týdně. Vydává se ve formě zkomprimovaného XML souboru, který má v době psaní této práce 29GB v komprimované podobě a 400GB v rozbalené podobě. Ten se dá získat buď přímo z webu OSM, či serverů partnerů projektu OSM a serverů třetích stran, které tento soubor šíří dále. Existuje i .torrent soubor pro stažení pomocí peer-to-peer protokolu BitTorrent[1], aby se serveru co nejvíce ulevilo. Z tohoto souboru pak dále existují tzv. extrakty, které jsou dělány firmami třetí strany a nejčastěji jsou to extrakty světadílů, potom států a někdy i jednotlivých krajů, či provincií státu.

Pokud je potřeba aktualizovat databázi založenou na datech z OSM častěji než jeden týden, využívá se tzv. diffů. Diff je soubor ve formátu OsmChange, který se vydává buď denně, každou hodinu nebo každou minutu. Tedy existují tři různé sady diffů. Diff obsahuje všechny změny do databáze, které proběhly od předchozího diffu. Tyto diffy opět můžeme najít v globálním měřítku pro celý svět, dostupné z webu OSM. Existují diffy i pro jednotlivé extrakty, ale ty jsou zdarma dostupné opět z webů firem třetích stran nebo partnerů projektu OSM. Pro českou republiku existují denní diffy z webu geofabrik.de, což je největší zprostředkovatel OSM dat v podobě extraktů. At'už planet.osm nebo jednotlivé extrakty jsou dostupné v různých formátech. Nejčastěji najdeme data ve formátu XML zkomprimované algoritmem bzip2 a potažmo v nezkomprimované XML podobě. Od tohoto formátu se postupně opouští a aktuální propagovaný formát je binární .pbf formát založený na formátu Protocol Buffers, který vymyslela společnost Google, Inc. A pak existují i další formáty, ale ty jsou dostupné jenom z webů třetích stran a samotné OSM je nepodporuje.

3.1 Čtení OSM souborů

V rámci této diplomové práce jsem napsal nástroj, který dokáže zpracovat a uložit data potřebné k algoritmům pro počítání trasy a časů dojezdu vozidel, dále pak data poštovních adres pro vyhledávání adres pomocí GPS souřadnic a naopak a dále pak body zájmů, jako jsou restaurace, zastávky, parkoviště atp. a to vše s verzováním a aktualizacemi. Tento nástroj jsem implementoval s možností, že by se v budoucnu mohl změnit, jak datový zdroj, tak datový cíl (pamět, databáze apod.). Proto je celý nástroj založený na abstrakcí tříd a všechny abstraktní třídy a rozhraní, které jsou dále rozšiřovány podle datového zdroje, či datového cíle, se nachází v knihovně Mapping.Data.Core. Jsou to ty nejzákladnější třídy, které tento nástroj využívá. Například, zde jsou třídy pro objekty pro routování jako uzly, cesty a graf a rozhraní pro DataSource objekty, DataTarget objekty,

[1] BitTorrent je protokol určený pro sdílení souborů mezi uživateli, který na rozdíl od klasického client-server stahování, daný soubor rozkouskuje a jednotlivé kousky se šíří mezi uživateli, kteří daný soubor sdílí nebo stahují a tím padém je přenos rychlejší a není zatěžován jeden server, ale zátěž je distribuována na všechny klienty.



Obrázek 2: Náhled na strukturu importeru

které říkají jaké metody musí objekty podporovat, aby z nich mohlo být čteno (DataSource), či ukládáno (DataTarget). Třídy Továren apod. V tento okamžik využívám v tomto nástroji jako zdroj dat pouze OSM a cíl, kam se data ukládají jsou dva, jeden je do paměti pro malé datové sady a potom do PostgreSQL databáze. Proto se dále v rámci této diplomové práce budu zmiňovat pouze o třídách, co se zabývají zdrojem dat z OSM a ukládají je do databáze PostgreSQL. Abstraktní návrh nástroje můžete vidět na obrázku 2.

Úplně první část mé práce bylo vůbec načíst soubory z OSM projektu. Jako první jsem v knihovně Mapping.Data.Osm.Types vytvořil třídy, které reprezentují jednotlivé elementy OSM, včetně verzování a podpory tagů, tak jak jsou popsány v sekci 2. Všechny elementy dědí z třídy OsmElement, která již obsahuje společné atributy všech elementů, jako jsou id, tagy, verze apod. navíc jsem přidal vlastnost ElementType, která je enum, říkající o jaký typ elementu se jedná, tento se při každé inicializaci zděděného objektu musí nastavit v konstruktoru třídy. Tato vlastnost je důležitá a umožňuje například dávat všechny elementy do jedné kolekce a poté je pomocí této vlastnosti přetypovat zpátky na příslušný element. Dále bylo nutné vytvořit třídy, které dokáží číst OSM soubory a z jednotlivých nejpoužívanějších formátů dokáží sestrojit třídní reprezentaci objektů, tzv. parsery. Pro čtení a následné zpracování jednotlivých objektů z OSM souborů do POCO (plain old CLR objects) tříd, které jsou dále využívaný v aplikacích, jsem vytvořil tři Parsery a na nich postavené Readery a potažmo DataSource třídy.

3.1.1 OsmXmlReader

První je OsmXmlReader v knihovně Mapping.Osm.Data.Xml, který umí číst a parsovat soubory v nezkomprimovaném XML formátu a využívá se ve třídě OsmXmlDataSource. Tento reader je rozšířenou implementací třídy XmlReader z .NET Frameworku a funguje na stejném principu jako XmlTextReader. Nejčastější použití je volání metody Read(), která, na rozdíl od klsického XmlTextReaderu, který parsuje xml elementy, se pokusí přečíst a parsovat další OSM element v souboru a vrátí true dokud soubor obsahuje OSM elementy a daří se je parsovat nebo false pokud narazí na konec souboru. Pokud metoda Read() vrátí true, můžeme z vlastnosti Element získat právě přečtený OSM element. Navíc umožňuje nastavit pomocí boolean vlastností IgnoreNodes, IgnoreWays, IgnoreRelations specifikovat, které OSM elementy se mají přeskočit, pokud na ně parser narazí a umožnit, tak rychlejší čtení souboru, pokud nás tyto OSM elementy nezajímají. Reader funguje tak, že podle jména XML elementu rozezná o jaký OSM element se jedná a ten se pokusí rozparsovat pomocí TryParse metod pro jednotlivé datové typy. Ukázku XML souboru, který reader čte můžete vidět ve výpisu 1. Všechny OSM soubory v XML formátu podle doporučení z dokumentace OSM projektu, by měly být řazeny tak, že nejprve jsou uzly, potom cesty a nakonec relace, ale nemusí to být pravda, proto toho faktu OsmXmlReader nevyužívá a parsuje vše nezávisle na pořadí.

```
<?xml version="1.0" encoding="UTF-8"?>
<osm version="0.6" generator="CGImap 0.0.2">
 <node id="298884269" lat="54.0901746" lon="12.2482632" user
    ="SvenHRO" uid="46882" visible="true" version="1"
    changeset="676636" timestamp="2008-09-21T21:37:45Z"/>
  <tag k="name" v="Neu Broderstorf"/>
 </node>
 <node id="298884272" lat="54.0901447" lon="12.2516513" user
    ="SvenHRO" uid="46882" visible="true" version="1"
    changeset="676636" timestamp="2008-09-21T21:37:45Z"/>
 <way id="26659127" user="Masch" uid="55988" visible="true"
    version="5" changeset="4142606" timestamp="2010-03-16
    T11:47:08Z">
  <nd ref="292403538"/>
  <nd ref="261728686"/>
  <tag k="highway" v="unclassified"/>
  <tag k="name" v="Pastower Strase"/>
 </way>
 <relation id="56688" user="kmvar" uid="56190" visible="true
    " version="28" changeset="6947637" timestamp="2011-01-12
    T14:23:49Z">
  <member type="node" ref="294942404" role=""/>
  <member type="node" ref="364933006" role=""/>
  <tag k="name" v="Kustenbus Linie 123"/>
  <tag k="operator" v="Regionalverkehr Kuste"/>
```

```
  <tag k="ref" v="123"/>
 </relation>
</osm>
```

Výpis 1: Ukázka xml dat z OSM

3.1.2 OsmCompressedXmlReader

Druhý reader, který jsem pro tento projekt vytvořil je reader, který umí číst XML formát v bzip2 zkomprimovaném formátu. Tento reader je přímo součástí OsmXmlReaderu a stačí specifikovat v konstruktoru o jaký proud (stream) dat se jedná a reader se už podle toho zařídí a stream si za běhu bude dekomprimovat, tudíž není ani třeba zkomprimované soubory před čtením dekomprimovat, reader si dekomprimuje soubor podle velikosti sliding window, takže i náročnost na paměť při dekomprimaci je velmi malá a otisk v paměti je pár kilobytů podle velikost sliding window použitého pří komprimaci souboru metodou bzip2.

3.1.3 OsmPbfReader

Jako třetí a poslední jsem vytvořil Reader, který dokáže číst OSM soubory v novém formátu PBF. OSM formát PBF je založený na Protocol Buffers od společnosti Google. Hlavní výhodou formátu je, že soubory jsou o 50 procent menší než XML zkomprimované gzip metodou a o 30 procent menší než XML zkomprimované metodou bzip2 a 5x rychleji se zapisují než xml v gzip formátu a měly by se číst 6x rychleji než XML v gzipu. V Googlu tento protokol vytvořili a hojně používají pro vzdálené volání procedur, právě jako náhradu za formát XML. Pro každý pbf soubor musí existovat externí specifikace, která popisuje strukturu objektů, tzv. messages v případě pbf. Na základě této specifikace se pomocí utility ProtoGen.exe, která je součástí knihovny protobuf-net od googlu vygenerují třídy, které reprezentují hierarchii objektů, jejich vlastností atd. Bez těchto tříd nelze pbf formátu porozumět, protože samotný soubor s binárními daty v sobě nenese metadata o struktuře objektů a serializaci. Do projektu jsem uložil obě dvě specifikace pbf formátu pro OSM data. První fileformat.proto specifikuje dva typy Blob a BlobHeader, který říká jak vypadá uložení dat na nejnižší úrovni, kde všechny data jsou uložena do 8 kB bloků tak, že vždy první 4 byty udávají kolik bytů má hlavička (BlobHeader) a BlobHeader dále popisuje data (blob), která následují hned za hlavičkou. Dále pak soubor osmfileformat.proto, který už specifikuje uložení jednotlivých elementů do blobů. Tento Reader byl v době psaní kódu první mě známá implementace tohoto readeru v jazyce C#, existoval pouze PbfReader v projektu OsmSharp, ale ten do nedávna nepodporoval nový pbf formát, který OSM vytvořilo a nahradilo tak starší, již nepoužívanou verzi.

3.1.3.1 Srovnání rychlosti čtení

Abych srovnal rychlost jednotlivých readerů a mohl vybrat nejvhodnější formát pro

import, rozhodl jsem se změřit čas čtení souboru v různých formátech. Vyzkoušel jsem všechny na stejném vzorku dat z OSM, čili na vzorku, který obsahoval úplně ty stejné data jenom zakódované do jiného formátu. Pro testování jsem si vybral extrakt pro Českou republiku ze serveru GeoFabrik.de, který nabízí extrakty ve všech třech formátech. Soubory měly v jednotlivých formátech velikosti jak můžete vidět v tabulce 1.

Jméno souboru	Formát	Velikost na disku
czech-republic-latest.osm	nekomprimované XML	7.15 GB
czech-republic-latest.osm.bz2	XML zkomprimované pomocí bzip2	496 MB
czech-republic-latest.osm.pbf	Protocolbuffer binární formát	319 MB

Tabulka 1: Velikosti formátů extraktů ČR

Každý z těchto souborů jsem četl k němu vytvořeným readerem, který pro každý element v souboru vytvářel příslušné POCO třídy a já měřil časy, jak dlouho trvalo každému readeru přečíst celý soubor od začátku až do konce. Výsledky můžete vidět v tabulce 2. Jenom pro srovnání jsem zde přidal i čas, za jak dlouho přečte nezkomprimované XML klasický XmlTextReader, který je součástí .NET Frameworku pro čtení XML souborů, v tomto případě jsem ovšem neporovnával ani jméno elementu, natož deserializoval jednotlivé OSM elementy, pouze volání metody Read() až do konce souboru. Je zde jenom pro srovnání rychlosti čtení XML souboru, pokud se objekty nedeserializují.

Jméno souboru	Typ readeru	Doba čtení celého souboru
czech-republic-latest.osm	XmlTextReader	2 minuty 56 sekund[2]
czech-republic-latest.osm	OsmXmlReader	4 minuty 17 sekund
czech-republic-latest.osm.bz2	OsmXmlReader (bzip2=true)	12 minut 16 sekund
czech-republic-latest.osm.pbf	OsmPbfBinaryReader	2 minuty 59 sekund

Tabulka 2: Rychlost čtení jednotlivých formátů

Jak je vidět z naměřených hodnot, je jasné, že pbf formát je pro čtení a samotný import nejlepší, protože čtení a parsování tohoto formátu proběhlo v téměř stejném čase jako přečtení XML formátu bez parsování. Navíc tento formát je nejlepší i co se do velikosti souboru týče, protože na disku zabírá nejméně místa, takže to, co se o tomto formátu povídá, je dokázáno i v praxi a je opravdu efektivnější a rychlejší, než ostatní formáty použité pro OSM data.

Pro jednoduchost použití těchto readerů v DataSource třídách, jsem vytvořil abstraktní třídu DataSourceFactory, která má jedinou abstraktní třídu CreateDataSource, která bere jako parametr cestu k souboru a sama podle typu souboru vrátí příslušnou implementaci DataSource. Tato třída je vytvořena podle návrhového vzoru Továrna (Factory) [3] a doporučuji všechny DataSource třídy inicializovat právě přes tuto Továrnu. V případě importu dat z OSM tedy potom v knihovně Mapping.Data.Osm najdeme třídu OsmDataSourceFactory, která mimo jiné, kromě samotné abstraktní třídy DataSourceFactory, implementuje návrhový vzor Singleton [3] a pomocí vlastnosti Instance, můžeme volat danou singleton instanci Továrny, která nám vytvoří konkrétní OsmDataSource. Příklad

použití můžete vidět ve výpise 2.

```
string filePath = @"c:\czech-republic-latest.osm.pbf";
var dataSource =
   OsmDataSourceFactory.Instance.CreateDataSource(filePath);
```

Výpis 2: Příklad použití DataSourceFactory

3.2 Selekce a indexace OSM dat

Dalším krokem importu dat z OSM bylo čtená data nějak interpretovat, vybrat potřebná data a ty potom přetransformovat do našeho formátu a uložit pro pozdější potřeby frameworku, tzv. zaindexovat. Tento krok se ukázal jako nejsložitější.

3.3 Použitý formát dat

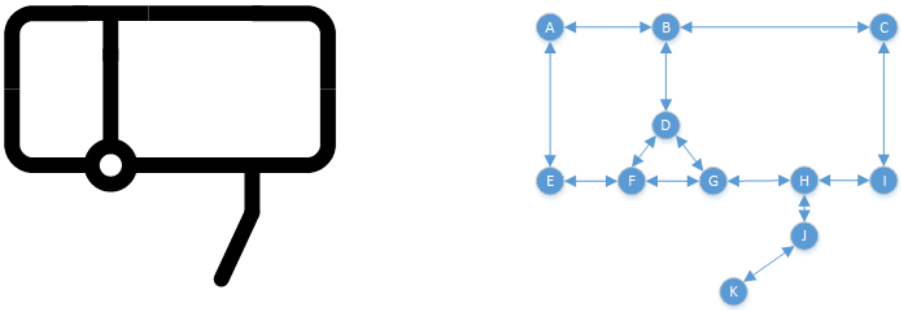
Nejprve bylo třeba navrhnout objektovou reprezentaci dat pro routovací algoritmy, jako Dijkstrův, Allstar a další. Routovací algoritmy používané v tomto připravovaném frameworku pracují s grafovou reprezentací cest, protože, jak můžete vidět na obrázku 3 na straně 21, každá cesta se ve skutečnosti dá reprezentovat jako orientovaný graf. Podle definice 3.1 Jiřího Sedláčka [14] je orientovaný graf, potažmo tedy silniční síť, množinou uzlů a k nim přiřazených orientovaných hran, tedy hran, které určují směr, jakým se dá přejít z uzlu do uzlu, které tyto hrany spojují.

Definice 3.1 *Orientovaný graf je dvojice* $G =< V, E >$*, kde:*

- V *je neprázdná množina tzv. vrcholů (uzlů)*
- *a* $E \subseteq V \times V$ *je množina uspořádaných dvojic vrcholů, tzv. (orientovaných) hran.*

Datové struktury pro grafy jsou známé především z backendů sociálních sítí, protože sociální síť, podobně jako silniční síť, tvoří graf uzlů, kde uzly jsou lidé (namísto křižovatek) a hrany jsou vazby mezi lidmi (namísto cest spojující křižovatky), proto jsem hledal motivaci v datových strukturách a uložištích právě sociálních sítí. Podle Lorenza Albertona [1] i Mohammada Beydouna a RamziA. Haratyho [2], kteří popisují jak ukládat grafy do relačních databází, existují celkem tři základní způsoby, jak objektově reprezentovat graf. Jsou to:

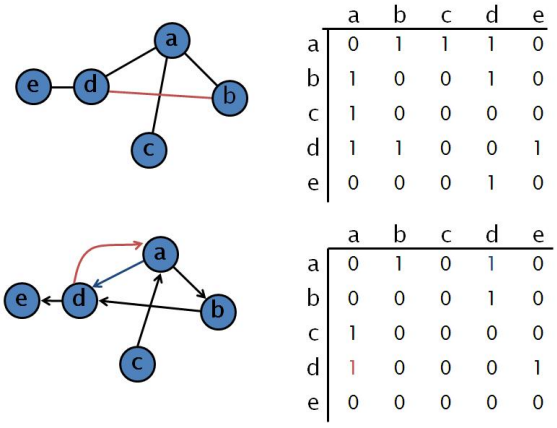
- Seznam přilehlosti
- Matice sousednosti
- Stromová struktura



Obrázek 3: Reprezentace cest jako orientovaný graf

Stromová struktura má dle nich [1] [2] výhody pokud se graf nemění, protože při vložení uzlu nebo přidání hrany, to znamená, že se celý strom musí přeskládat a proto se v praxi téměř nepoužívá, hodí se jenom pro speciální případy grafů. Tento typ uložení je navíc velmi komplikovaný v případě, že jej chceme uložit do relační databáze, což byl můj případ. Proto jsem tento typ uložení vyloučil.

Matice sousednosti je velmi jednoduchou strukturou a velmi používanou matematiky. Graf se pomocí matice sousednosti reprezentuje jako čtvercová matice n-tého stupně(viz. Definice 3.3), kde n je rovno počtu uzlů v grafu. Orientovaný graf se pomocí této matice potom reprezentuje tak, že uzly jsou ve stejném pořadí reprezentovány jako řádky a sloupce a pokud existuje hrana z uzlu A do uzlu B, nastaví se na řádku uzlu A v sloupci uzlu B hodnota na true. Nejlépe opět vysvětleno v obrázku 4.



Obrázek 4: Matice sousednosti jako reprezentace grafu a orientovaného grafu. Zdroj: `http://sourcecodemania.com/graph-implementation-in-cpp/`

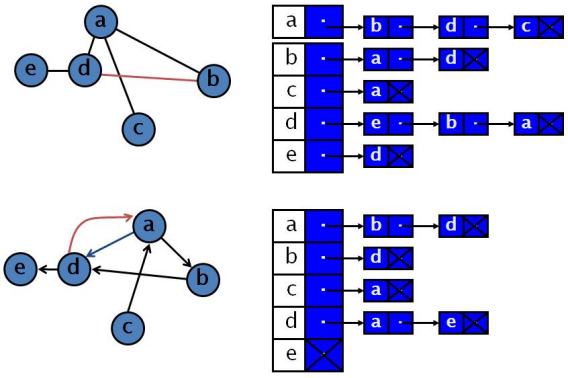
Tento způsob je náročnější na pamět, či místo na disku než v případě seznamu přilehlosti. Velmi rychle se v něm zjišťuje, zda mezi uzly existuje hrana a hodí se pro husté

grafy[3], což není případ grafu silniční sítě, kde uzly (křižovatky) mají průměrně 3-4 hrany. Tuto reprezentaci jsem ze začátku používal pro reprezentaci silniční sítě malých oblastí, ale pro větší území je tento způsob nevhodný, proto jsem jej nakonec nepoužil. Navíc při přechodu, z uložiště RAM pamětí do relační databáze se projevil tento způsob reprezentace jako nepoužitelný, neboť při každém přidání uzlu by se musela tabulka ukládající uzly předělat a přidat sloupec a to vyžaduje DDL operaci, která je velmi drahá co se týká výpočetního výkonu. Tuto reprezentaci si, ale dokážu představit při použití pro NoSQL databáze.

Definice 3.2 *Maticí typu (m,n) nazýváme schéma objektů sestavených do m řádků n sloupců.*

Definice 3.3 *Čtvercová matice n-tého stupně je taková matice, která má počet řádků i sloupců roven n.*

Seznam přilehlosti funguje, tak, že každý uzel si udržuje seznam uzlů, které jsou mu přilehlé, neboli seznam uzlů, do kterých existuje hrana z daného uzlu viz. obrázek 5.



Obrázek 5: Graf reprezentovaný seznamem přilehlosti. Zdroj: `http://sourcecodemania.com/graph-implementation-in-cpp/`

Tento způsob, má menší požadavky na paměť, než matice sousednosti a ukázalo se, že je velmi vhodný pro routovací algoritmy, protože ty procházejí uzel po uzlu a zjišťují, kam se z něj mohou dostat, což v tomto případě znamená jednoduché procházení seznamu. Tento způsob uložení grafu jsem mírně modifikoval, abych mohl k hranám přidat doplňující informace nutné pro routovací algoritmy, tak že místo seznamu uzlů, jsem uzlům přidal odkazy na hrany, které v sobě mají informace i o uzlech, ale princip uložení grafu zůstává ve své podstatě pořád stejný jako původní seznam přilehlosti. Jako reprezentaci uzlu orientovaného grafu, neboli křižovatky, jsem vytvořil třídu Node, která má vlastnosti, jaké můžete vidět v tabulce 3. Pro reprezentaci zeměpisné délky a šířky jsme v rámci celého frameworku vybrali E6 formát, který se dá reprezentovat 32bitovám integerem, z důvodu jeho malé paměťové náročnosti oproti klasickému formátu ve stupních, který se

[3]Hustý graf je takový graf, ve kterém mají uzly mnoho hran. Například pokud bysme vzali graf o 4 uzlech, tak hustý graf by byl graf, kde by existovala hrana mezi každýma dvěma uzly.

ukládá do 64bitového datového typu double, který má dvakrát větší velikost. E6 formát je ve své podstatě uložení zeměpisné délky, či šířky v mikrostupních, protože se získá, tak, že například zeměpisnou šířku zapsanou ve stupních získáme vynásobením 10^6 a zpátky opět vydělením. Příklad převodu v jazyce C# můžete vidět ve výpise 3. Samozřejmě kvůli faktu, že souřadnice je takto uložena v nejvyšší přesnosti na mikrostupně, jakákoliv vetší přesnost se převodem do E6 formátu ztratí. Přesnost na cm formátu E6 je podle výpočtů Gordon Carrieho [4] 11 cm, což je pro potřeby frameworku dostačující přesnost.

Jméno vlastnosti	Typ vlastnosti	Popis vlastnosti
Id	long	jednoznačný identifikátor uzlu
Lat	int32	Zeměpisná šířka uložena v E6 formátu (v mikrostupních)
Lon	int32	Zeměpisná délka uložena v E6 formátu (v mikrostupních)
Edges	List<Edge>	Seznam všech vstupujících i vystupujících hran

Tabulka 3: Vlastnosti třídy Node, reprezentující uzel v orientovaném grafu

```
int latitudeE6 = int(49,5766371488571 * 1000000)
double latitude = latitudeE6 / 1000000d
```

Výpis 3: Převod E6 formátu souřadnic na stupně a zpět. Ukázka v jazyce C#

Pro objektovou strukturu orientovaných hran, reprezentující silnice, jsem vytvořil třídu Edge, která má vlastnosti, které můžete vidět v tabulce 4. V případě hrany (třídy Edge) nebylo třeba použít jednoznačné id, protože hranu jednoznačně určují dva uzly, která hrana spojuje. U orientované hrany, je důležité si uvědomit, že záleží na pořadí uzlů. Proto uzel označený vlastnosti Id, je vždy uzel první a vlastností ToNodeId je označený druhý uzel. Pro jednoduchost jim říkejme uzel A a uzel B. Díky tomu, že máme takto označené pořadí uzlů, můžeme použít vlastnosti IsForward a IsBackward, které tak určují v jakém směru je silnice (hrana) sjízdná. Pokud je vlastnost IsForward nastavena jako true, můžeme po hraně přejít z uzlu A do uzlu B, pokud je false nelze ji v tomto směru přejít. V případě vlastnosti IsBackward to platí obráceně, pokud je vlastnost nastavena na true, znamená to, že lze přejít z uzlu B do uzlu A a pokud je nastavena na false, znamená to, že nemůžeme přejít z uzlu B do uzlu A. Pokud jsou obě vlastnosti nastaveny na true, lze hranu přecházet z uzlu A do uzlu B i obráceně z uzlu B do uzlu A. Vlastnost Geometry, může být v případě hran pouze linie nebo čára, chcete-li. Tato vlastnost je ve výchozím momentě, při sestrojení hrany pouze přímá linie z uzlu A do uzlu B. Nepřímá, geometricky definovaná linie se z ní může stát až po minimalizaci grafu. O minimalizaci se dočtete více v následující kapitole o minimalizaci. Vlastnost Geometry se využívá hlavně pro vykreslování cesty do bitmapy a následném zobrazení v mapě.

Pro objektovou reprezentaci bodů zájmu slouží třída Poi (Point of interest). Její vlastnosti můžete vidět v tabulce 5. Z OSM dat získáváme celkem 140 typů bodů zájmů. Pod typem si můžete představit, třeba restaurce, nemocnice, kostel, parkoviště atp., které jsem

Jméno vlastnosti	Typ vlastnosti	Popis vlastnosti
Id	long	id uzlu, ve kterém hrana začíná
ToNodeId	long	id uzlu, do kterého hrana směřuje
IsForward	bool	Boolovská proměnná říkající, zda jde přejít hranu v přímém směru
IsBackward	bool	Boolovska proměnná říkající, zda jde přejít hranu v opačném směru
SpeedKmh	byte	maximální rychlost po dané hraně (silnici)
LengthM	uint	délka hrany v metrech
RoadClass	ERoadClass	Enum specifikující o jakou třídu silnice se jedná. (dálnice, pro motorová vozidla, 1.třídy atp.
VehicleAcces	EVehicleAccess	Enum specifikující jaké typy vozidel mohou po silnici jezdit
Lanes	byte	počet jízdních pruhů
Geometry	LineString	Vlastnost, která má uloženou geometrické vlastnosti cesty

Tabulka 4: Vlastnosti třídy Edge, reprezentující hranu v orientovaném grafu silniční sítě

rozdělil do celkem deseti kategorií. Kategorie bodu zájmu může být jedna z těchto: jídlo a pití, vzdělávání, doprava, finance, zdravotní péče, zábava a kultura, turistika, lokace nebo ostatní. Pro umístění na mapě slouží vlastnost Location, která udává střed bodu zájmu. Pro vykreslení na mapu slouží vlastnost Shape, která udává geometrii, která může být buď bod, linie nebo polygon. Pokud v OSM datech existuje i poštovní adresa daného bodu zájmu, je uložena ve vlastnosti Address, v opačném případě je null. Pro objektovou reprezentaci poštovních adres slouží tedy třída Address. Třída Address obsahuje informace o státě, městě, ulici, čísle popisném nebo evidenčním a směrovacím čísle. V OSM datech je téměř pravidlem, že adresa není úplná, proto ikdyž samotná vlastnost Address u objektu Poi není null, tak to neznamená, že objekt adresy je úplný. Běžně chybí například stát.

3.4 Uložiště zpracovaných dat

Pro potřeby frameworku, jsme se rozhodli jako utložiště dat použít relační databázi PostgreSQL ve verzi 9, navíc s rozšířením o geometrie pomocí PostGis pluginu ve verzi 2.0. Pro toto uložiště jsme se rozhodli z následujících důvodů:

1. Toto uložiště ve frameworku již používáme v rámci jiného projektu.

2. Databázový engine PostgreSQL je velmi vyspělý engine a mnoho práce, kterou bych jinak v projektu musel dělat je v databázi již implementována. (indexace, třídění, vyhledávání apod.)

Jméno vlastnosti	Typ vlastnosti	Popis vlastnosti
Id	long	jednoznačný identifikátor bodu zájmu
Name	string	Zobrazované jméno bodu zájmu
Description	string	Krátký popis bodu zájmu
Location	GeoPoint	Gps souřadnice bodu zájmu
Shape	Geometry	Geometrický tvar bodu zájmu
Category	EPoiCategory	Enum říkající do jaké kategorie bod zájmu spadá (celkem 10)
Type	EPoiType	Enum říkající o jaký typ bodu zájmu se jedná (celkem 140)
Address	Address	Poštovní adresa bodu zájmu

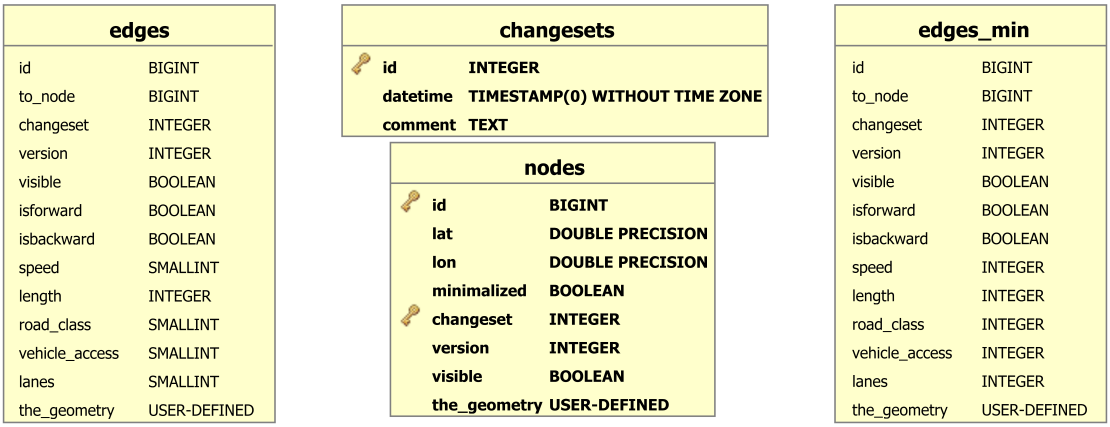
Tabulka 5: Vlastnosti třídy Poi, reprezentující bod zájmu

3. Díky rozšíření PostGis 2.0 jsou v databázi k dispozici i geografické funkce, které v práci hojně využívám. Jsou to například funkce pro výpočty vzdálenosti mezi jednotlivými body, porovnávání polygonů, funkce na určení zda bod leží v polygonu, spojování geometrií, vytváření indexů pro geometrie. Operace spojování bodů do geometrií a počítání vzdálenosti. Všechny tyto geometrické operace by šly samozřejmě dělat v samotném nástroji na import, v paměti programu a databázi pak použít jenom čistě jako uložiště, ale kvůli struktuře vyexportovaných dat z OSM, by bylo potřeba velké množství RAM paměti, proto se jako lepší varianta ukázalo použít právě funkce databáze. Více o tomto problému v sekci 3.7.

4. I programy mimo tento framework, budou schopné číst přímo z této databáze data pomocí jazyka SQL. Takovým příkladem jsou třeba vizualizační programy, které umí již s PostGis datovým formátem pracovat, tudíž je jednoduché si třeba zobrazit silniční síť, čehož jsem hodně využíval pro ověřování zpracovaných dat.

5. Databáze i rozšíření PostGis jsou zcela zdarma a denně se používají ve velkých projektech jako je Foursquare, samotný projekt OSM a další, proto se předpokládá, že databáze a její funkce budou velmi odladěné právě pro naši potřebu, jelikož se těmto projektům velmi podobáme svým zaměřením.

Pro uložení dat jsem v databázi vytvořil tři na sobě nezávislé skupiny tabulek, každá sloužící pro jiný účel. První skupina jsou tabulky, ve kterých jsou uloženy data pro routing. UML diagram tabulek můžete vidět na obrázku 6 Jsou to tabulky nodes, kam se ukládají uzly, edges pro hrany, edges_min pro zminimalizované hrany a changeset pro uložení informací o provedených změnách. Sloupce u jednotlivých tabulek jsou vypovídající samy o sobě a v podstatě kopírují vlastnosti tříd, které jsem popsal výše, proto je zde nebudu podrobně popisovat. Jelikož, jeden z požadavků na uložiště bylo aby se ukládaly i starší verze dat a bylo možné vrátit případné změny do databáze, jsou v tabulce uzlů i hran sloupce určené pouze pro verzování. Verzování funguje obdobně jako u projektu



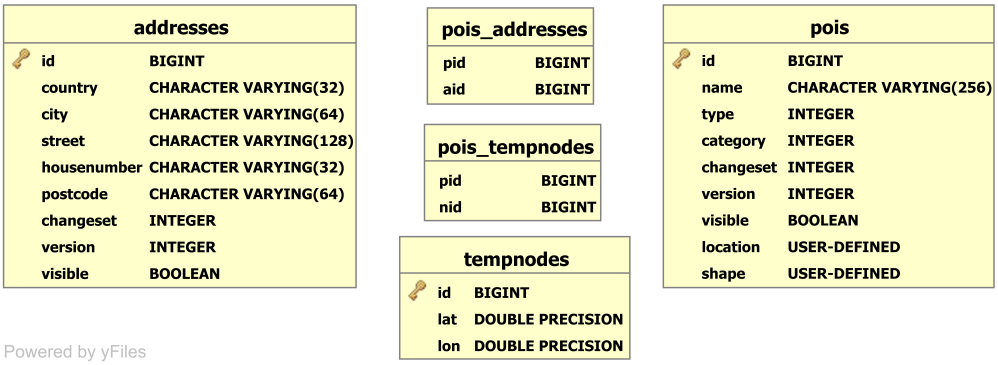
Obrázek 6: UML Diagram z použité PostgreSQL databáze

OSM. Do databáze by se mělo zasahovat pouze přes třídu PgDatabase v knihovně Mapping.Data.PostGis.dll, která se stará o verzování. Před každým zásahem do databáze, který zahrnuje přidání, smazání nebo aktualizaci objektu v databázi, je třeba přes tuto třídu zavoláním metody NewChangeset vytvořit nový objekt typu ChangeSet, který bude mít z databáze přidělené id. Pro vytvoření je třeba jako parametr metodě předat komentář k changesetu, například ”aktualizace dat z OSM”. Třída PgDatabase potom každou provedenou změnu do databáze spojí právě s aktuálně vytvořeným Changesetem. Třída se automaticky stará o verze objektů. Přes třídu PgDatabase nejde fyzicky, žádný objekt z databáze smazat, protože místo fyzického smazání se u objektu nastaví vlastnost visible na false, aby se případně dalo ke smazanému objektu vrátit.

Poznámka 3.1 U tabulky edges_min chybí tyto atributy, protože tabulka se generuje z posledních verzí objektů v tabulce edges a pokud se změní tabulka edges, tak tabulka edges_min se vytvoří znova. Sloupec the_geometry je typu Geometry, což je datový typ právě z pluginu PostGis.

Druhá skupina tabulek jsou tabulky sloužící k uložení bodů zájmů a poštovních adres. UML diagram můžete vidět na obrázku 7. Sloupce, stejně jako u tabulek pro routing, odpovídají vlastnostem tříd reprezentující dané objekty, tedy třídám Poi a Address, proto nemá cenu je znovu popisovat. U tabulek pois a addresses opět můžeme vidět sloupce visible, version a changeset sloužící k verzování objektů. Stejná pravidla zacházení jako s tabulkami pro routing platí i zde, tedy pro práci používejte vždy třídu PgDatabase, která se stará o verzování.

Poznámka 3.2 Sloupec location a shape v tabulce pois jsou datového typu Geometry z pluginu PostGis.



Obrázek 7: UML diagram pro tabulky bodů zájmů a poštovních adres

Třetí skupina tabulek slouží k uložení surových OSM dat, kvůli cachování. Cache surových dat je popsána dále v textu v kapitole 3.6 na straně 35.

3.5 Interpretace OSM dat

Velmi důležitou částí celého importu dat je interpretování a porozumění OSM dat pro převod do našeho formátu(viz kapitola 3.3), který používají ostatní části frameworku. Pro tento účel slouží třídy typu Interpreter, které se snaží zjistit o jaký typ dat se jedná a ty transformovat do požadovaného formátu. Rozpoznávání dat je umožněno hlavně díky systému štítku, který slouží v OSM k popisování dat. Když uživatel vkládá data do projektu OSM, může data označit libovolným počtem a druhem štítků. To by ovšem znamenalo, že každý si do projektu vloží data a popíše si je podle sebe a k ničemu by to nevedlo, protože by ostatní nevěděli, co dané štítky znamenají a jaký popisují objekt. Proto v OSM existují tzv. vlastnosti dat (z angl. features), konkrétně se jim říká mapové a navržené vlastnosti[4] dat. Tyto vlastnosti jsou laicky řečeno doporučení, jak označovat data pomocí štítků tak, aby je ostatní uživatele poznali a byli schopni použít. Rozdíl v mapových a navržených vlastnostech je potom ten, že způsoby označování definované v mapových vlastnostech, jsou způsoby, které jsou v OSM považované za standart. Navržené vlastnosti jsou způsoby označování dat, které navrhují uživatelé OSM pro nový typ dat nebo jako vylepšený způsob označování dat, která je již definovaný v mapových vlastnostech. Pokud komunita a představitelé projektu OSM rozhodnou, že nějaký z navržených způsobů označování je lepší než již existující, může jej v mapových vlastnostech nahradit nebo pokud popisuje zcela nový typ dat, která jsou populární, může se takto rovněž dostat do mapových vlastností. Dále zde ještě existují tzv. zastaralé vlastnosti[5], které obsahují staré způsoby označování, které byly nahrazeny novými, jelikož už nejsou dostačující. Data označována nějakým způsobem ze zastaralých vlastností, jsou postupně

[4]Mapové a navržené vlastnosti jsou můj doslovný překlad. V originále se jmenují Map features a Proposed features

[5]Opět se jedná o můj doslovný překlad do češtiny. V originále je to Depricated features

komunitou přeznačována na způsob nový.
Způsob označování štítky v realitě znamená, že daný OSM element musí obsahovat štítky s určitou kombinaci klíčů a k nim přiřazených hodnot, aby mohl být považovaný za popisovaný objekt. Například v mapových vlastnostech existuje způsob jak označit dálnici a ten říká, že OSM element musí být typu Way (z kapitoly 2.1) a musí obsahovat štítek, který má klíč "highway"s hodnotou "motorway". Podobně jsou popsány další typy cest a obecně všechny objekty v OSM. Navíc pro většinu způsobů "štítkování"existuje na webu wiki.openstreetmap.org stránka, popisující buď konkrétní klíč, nebo hodnotu klíče u použitého štítku, kde je uvedeno, jak můžou být označené doplňující informace. Například v případě zmíněné dálnice, tato stránka uvádí, že může být použit v kombinaci se štítkem name, který udává jméno silnice nebo štítkem ref, který udává referenční číslo silnice apod. Potom velmi dobrým nástrojem na hledání nejčastěji použitých kombinací štítků, prohlížení hodnot a obecnou statistiku štítků, slouží stránka taginfo.openstreetmap.org, kde jsou statistiky o jednotlivých štítcích, jaké hodnoty a klíče se vyskytují nejčastěji apod..
Hlavní problém takto otevřeného systému označování dat je, že všechny tyto způsoby označování dat jsou pouze doporučené, tedy není povinné je dodržovat a neexistují ani žádná opatření na straně OSM projektu, která by označování kontrolovala. Z tohoto důvodu a taky faktu, že klíče i hodnoty u štítků jsou typu string, se můžeme u dat setkat s nevalidními i nesmyslnými hodnotami. Například často se to stává u definice rychlosti, kde v doporučení je řečeno, že se označuje štítkem s klíčem speed, ale nikde už není řečeno, jaká jednotka rychlosti je použita (předpokládá se kilometry za hodinu), proto se objevují hodnoty "90", "30", "100 kmh"nebo třeba "50mph"či dokonce "10 uzlů"nebo "rychlost závratná". A tento problém se objevuje ne jenom u štítků, které si uživatelé sami vymyslí, ale právě i u štítků, které jsou definované v mapových vlastnostech. Není to problém jenom štítků samotných, výjimečně se totiž může stát i to, že jsou zaměněné hodnoty zeměpisné šířky a zeměpisné délky. Proto je velmi důležité důkladně validovat vstup a při převádění do hodnotových datových typů je nutné, v případě jazyka c#, používat metody TryParse[6] namísto Parse, protože je velmi pravděpodobné, že vstup do metody nebude validní. Pro interpretaci a validaci dat jsem v nástroji vytvořil tyto tři intepretery dat.

3.5.1 Interpreter pro polygony administrativních území

První z nich je třída OsmBoundaryInterpreter. Tato třída slouží v nástroji k interpretaci dat, která tvoří geometrické vyjádření administrativních území, jako jsou státy a města. Tento interpreter se používá při vytváření polygonů hranic měst a států, které se dále používají při dopočítávání maximální povolené rychlosti pokud v OSM datech chybí nebo není validní. Tento interpreter má tři metody:

[6]Metoda TryParse je dostupná u všech hodnotových datových typů v .NET Frameworku, která převádí textovou reprezentaci na daný hodnotový datový typ. Výhoda oproti metodě Parse je, že při neúspěšném převodu nenastane výjimka a běh programu je mnohem rychlejší. Metody Parse by se měly používat jen tehdy, pokud jsme si jistí, že vstup je validní.

1. **IsMunicipality**, která vrací true pokud daný OsmElement představuje polygon města nebo obecně administrativní území 2. úrovně, kam například v České republice spadají i obce a vesnice.

2. **IsCountry**, která vrací true pokud daný OsmElement představuje polygon pro hranici státu. Funguje i pro skupiny státu jako jsou třeba Spojené státy Americké, kde vrátí true pokud element představuje polygon pro hranici státu jako celku, nikoliv jednotlivých států USA.

3. **GetName**, která vrátí jméno daného města, či státu.

Funkce IsMunicipality Tato funkce k rozhodnutí, zda se jedná o město, využívá vlastnosti OSM dat pro obecné hranice (Map feature Boundaries). Označování hranic se v nedávné době změnilo na nový způsob a proto tato funkce umí poznat, jak mezi starým způsobem, který je již označený jako zastaralý, ale stále se většina dat vyskytuje právě v tomto formátu, tak novým způsobem značení. Tato funkce vrátí true tedy ve dvou případech:

1. Pokud element obsahuje štítek. který má klíč boundary s hodnotou administrative a zároveň štítek, který má klíč admin_level s hodnotou 8

2. Pokud element obsahuje štítek, který má klíč type s hodnotou boundary nebo multipolygon a zároveň štítek, který má klíč admin_level s hodnotou 8.

Funkce IsCountry Tato funkce k rozhodnutí, zda se jedná o stát, využívá stejné vlastnosti OSM dat jako funkce IsMunicipality, jenom s tím rozdílem, že v případě státu je administrativní úroveň 2. Tedy funkce vrátí true stejně jako funkce IsMunicipality jenom s tím rozdílem, že u štítku s klíčem admin_level musí být 2 ne 8, jak tomu bylo v případě funkce IsMunicipality.

Funkce GetName Tato funkce vrátí hodnotu štítku s klíčem name. Pokud štítek s klíčem name neexistuje vrátí prázdný řetězec.

3.5.2 Intepreter pro silniční síť

Další interpreter je třída OsmRoutingInterpreter, která, jak z názvu vyplývá, slouží k interpretaci OSM dat pro routovací algoritmy a pomocí ní se vytváří objektové modely reprezentující graf silniční sítě, které byly popsány v kapitole 3.3. Tento interpreter využívá mapové vlastnosti pro cesty (map feature highway). Tato třída má následující metody.

Funkce IsRoutable Tato metoda má návratovou hodnotu typu boolean a slouží k zjištění, zda se po daném elementu dá navigovat. V OSM neexistuje zaručený způsob, jak zjistit, zda se jedná o cestu. Tato metoda funguje, tak že vrátí true pokud daný element

obsahuje štítek s klíčem highway, který označuje elementy, které se obecně týkají cest, ale můžou jim být označeny například i dopravní značky, zastávky, odpočívadla a další objekty, které nejsou pro routing potřebné. Proto je třeba komplexnější kontrola, zda se jedná o cestu. To lze vyčíst z typu elementu a dalších vlastností, které tento interpreter může zjistit. Dá se říct, že tato metoda co nejrychleji zjistí, zda se má nástroj při importu tímto elementem vůbec zabývat, co se routingu týče.

Funkce IsOneWay Tato funkce má návratovou hodnotu typu nullable[7] boolean, která vrací true pokud je cesta jednosměrná v daném směru, false pokud je jednosměrná v opačném směru a null, pokud je obousměrná. Tato metoda využívá štítku s klíčem oneway, který udává zda se jedná o jednosměrnou cestu nebo ne. Pokud není tento štítek přítomný snaží se podívat na třídu silnice a pokud se jedná o dálnici nebo dálniční nájezd nebo sjezd, je jasné, že cesta je jednosměrná a dále se snaží zjistit zda se nejedná o kruhový objezd, protože zde je směr rovněž udán jasně.

Funkce GetMaxSpeed Tato funkce se pokusí zjistit maximální povolenou rychlost na silnici. Jak jsem již zmiňoval v úvodu interpretace dat, u rychlosti je problém s definicí jednotky. Maximální rychlost dle doporučení z mapových vlastností určuje štítek s klíčem maxspeed. Jeho hodnota může být čistě numerická, v tom případě se jedná o rychlost v kilometrech za hodinu. Dále se zde pak může uvést rychlost i s jednotkou. Pro metodu na validaci vstupu jsem vytvořil tento regulární výraz:

```
^([0-9]+) *(kmh|km/h|mph|kph)?$
```

tento regulární výraz splní řetězce začínající libovolně dlouhým číslem, následovaným libovolným počtem mezer a končící jedním nebo žádným z těchto čtyř zápisů jednotek rychlosti: kmh, km/h, mph a kph. Pokud jej řetězec splní metoda dále převede rychlost podle jednotky do rychlosti v kilometrech za hodinu a ověří, zda je rychlost v rozmezí 5 - 200 kmh, protože jiné hodnoty nejsou pro značení na silnicích validní. Pokud se rychlost nepodaří určit, metoda vrátí hodnotu 0. V nástroji je rovněž možnost nechat před samotným importem z OSM dat vygenerovat polygony měst a poté podle toho zda silnice leží ve městě se dá dopočítat výchozí rychlost. Více o polygonech měst v následující kapitole.

Funkce GetRoadClass Tato funkce vrací enum ERoadClass, kterým se označují třídy silnic. Enum ERoadClass může nabývat tyto hodnoty:

- **Motorway** pokud se jedná o dálnici
- **Primary** pokud se jedná o silnice 1. třídy

[7] Nullable je generická třída, která je součástí .NET Frameworku od verze 2.0, která funguje jako kontejner pro hodnotové datové typy, kterým je potom možné přiřadit hodnotu NULL, jako by to byl referenční datový typ.

- **Secondary** pokud se jedná o silnice 2. třídy
- **Tertiary** pokud se jedná o vedlejší silnice
- **Road** pokud se jedná o jinou cestu. Například. příjezdová cesta k rezidencím, servisní cesta, nespecifikovaná cesta, nebo například cesta v centru města, která je sdílená s chodci, obytná zóna apod.
- **None** pokud se nejedná o sjízdnou cestu

Funkce toto dokáže poznat z hodnot štítku s klíčem highway, které blíže specifikují o jaký objekt se jedná. Rozpoznává celkem 63 různých objektů a podle toho určuje, silniční třídu. Pokud objekt nerozpozná a přesto má štítek s klíčem highway, tak přiřadí cestě třídu None a hodnotu zapíše do Logu jako Warning, aby se později mohla přidat k rozpoznání.

Funkce GetNumOfLanes Tato funkce zjišťuje počet jízdních pruhů dané silnice, snaží se to zjistit z štítku lanes. Pokud daný štítek neexistuje přiřadí cestě výchozí hodnoty podle třídy silnice a podle toho, zda se jedná o jednosměrnou silnici nebo obousměrnou. Výchozí hodnoty jsou 2 pruhy pro dálnici v jednom směru (4 dohromady v obou směrech), 2 pruhy pro obousměrnou silnici 1. třídy a menší a 1 pruh pro jednosměrnou silnici.

Funkce GeVehicleAccess Tato funkce vrací enum EVehicleAccess, který říká, jaký typ vozidel smí po silnici jezdit. Enum EVehicleAcces může aktuálně nabývat těchto hodnot:

- **None** pokud je na silnici zakázán vjezd motorovým vozidlům
- **Truck** pokud po silnici smí jezdit nákladní motorová vozidla
- **PublicService** silnice pouze na speciální povolení
- **Regular** pokud je po silnici umožněno jezdit běžným motorovým vozidlům
- **All** toto je speciální hodnota, která nastaví EVehicleAccess pro všechny výše zmíněné hodnoty, kromě None.

Rozpoznávání na základě kombinace 36 různých štítků, jako jsou access, vehicle, motor_vehicle, motorcar, psv, hgv a další, které můžou indikovat povolení vjezdů výše zmíněným typům vozidel.

Funkce GetRef Tato funkce slouží ke zjištění referenčního jména silnice, jako jsou například D1, E55 a podobně.

Funkce GetName Funkce na zjištění zobrazovaného jména silnice, pokud není, vrátí hodnotu GetRef, pokud ani to neexistuje, vrátí prázdný řetězec.

3.5.3 Iterpreter pro body zájmů a poštovních adres

Interpreter pro body zájmů je implementovaný v tříde OsmPoiInterpreter. Tento interpreter slouží k interpretaci OSM dat pro převod do tříd Poi a Address.

Funkce GetType Tato funkce se ze štítků snaží zjistit o jaký typ bodu zájmu se jedná. Celkem rozpoznává přes 160 typů bodů zájmů. Konkrétně tato funkce ze štítků umí rozpoznat tyto body zájmu: AlpineHut, Artwork, Attraction, Appartment, Camping, CampingCaravan, Chalet, GuestHouse, Hotel, Hostel, Motel, Museum, PicnicSite, ThemePark, TouristInfo, Viewpoint, WildernessHut, Zoo, City, Town, Village, Hamlet, IsolatedDweling, Farm, Suburb, Neighbourhood, Continent, Country, County, Island, Islet, Locality, Region, State, GeneralLocationWithAddress, Bar, Bbq, BeerGarden, Cafe, DrinkingWater, FastFood, FoodCourt, IceCream, Pub, Restaurant, College, KinderGarden, Library, School, University, BicycleParking, BicycleRental, BoatRental, BusStation, BusStop, CarRental, CarSharing, CarWash, EvCharging, FerryTerminal, Fuel, GritBin, MotorcycleParking, Parking, ParkingEntrance, Taxi, SmallTrainStation, TrainStation, SubwayEntrance, TramStop, Aerodrom, AerodromTerminal, Atm, Bank, CurrencyExchange, BabyWatch, Clinic, Dentist, Doctors, Hospital, NursingHome, Pharmacy, RetirementHome, SocialFacility, Spa, Veterinary, ArtsCentre, Casino, Cinema, CommunityCenter, Fountain, NightClub, SocialCentre, StripClub, Studio, SwingersClub, Theatre, AirCompressor, AnimalBoarding, AnimalShelter, Bench, Brothel, BoatStorage, Clock, CourtHouse, Crematorium, Cemetery, Crypt, DrivingSchool, Embassy, EmergencyPhone, FireHydrant, FireStation, Gym, HuntingStand, MarketPlace, Monastery, WorshipPlace, Police, PostBox, PostOffice, Prison, PublicBuilding, Recycling, Sauna, Shelter, Shower, Telephone, Toilet, TownHall, VendingMachine, WasteBin, DrinkingWaterAnimal

Funkce GetCategory Pomocí této funkce se bod zájmu zařadí do kategorie. Kategorie se určuje na základě zjištěného typu z funkce GetType. Existují tyto kategorie bodu zájmů:

- **None** tato je speciální kategorie, která se používá jenom pro případ, že kategorie nebyla ještě metodou GetCategory určena
- **EatAndDrink** pro body zájmu týkající se jídla a pití
- **Education** pro všechny typy co mají něco společného se vzděláním
- **Transportation** pro všechny typy co se týkají dopravy
- **Finance** pro typy týkající se financí
- **HealtCare** pro typy okolo péče o zdraví (i zvířat)
- **EntertainmenArtsCulture** pro všechny typy co mají něco společného s uměním, zábavou a kulturou
- **Tourism** pro turistické body zájmu

- **Location** obecná kategorie pro typy jako stát, město, vesnice nebo např budovy bez speciálního využití s poštovní adresou atp.
- **Other** pro všechny body zájmu, co nešly zařadit do kategorií zmíněných výše.

Funkce GetAddress Tato funkce se pokusí získat ze štítků poštovní adresu daného elementu. Tato funkce zjišťuje číslo domu, ulici, město, zemi a poštovní směrovací číslo. V případě, že se alespoň jedna tato část adresy podaří zjistit, metoda vytvoří objekt address a vrátí jej. Pokud se nepodaří zjistit, žádnou část adresy funkce vrací null.

Funkce IsPoi Tato metoda má jako návratovou hodnoty typ boolean a slouží ke zjištění zda se jedná o bod zájmu, který má buď adresu, nebo se mu dá přiřadit konkrétní typ funkcí GetType, pokud ano vrátí true, jinak false.

3.6 Zpracování v paměti versus v databázi

Původní plán na zpracování dat bylo číst surová data z OSM, ty hned zpracovat a uložit do databáze a ty tak můžou být hned použité pro routovací algoritmy a další aplikace. Tento způsob se ukázal jako velmi problémový a funguje jenom pokud importujeme malý vzorek dat jako je, v případě pro Českou republiku, třeba kraj nebo pokud máme k dispozici dostatečné množství RAM paměti, které ovšem nejde dopředu spolehlivě určit. Je to dáno formátem OSM dat popsaných v sekci 2.1 a jejich roztříštěností. Podívejme se na tento problém v příkladu 3.1:

Příklad 3.1
Mějme jako vzorek dat extrakt z OSM v libovolném formátu (pbf, osm nebo osm.bz2) pouze pro Českou republiku. A chtějme zde, pro jednoduchost, získat pouze jeden polygon reprezentující hranici státu. Čili potřebujeme získat seřazený seznam GPS souřadnic, tak jak jdou v polygonu po sobě a z něj sestrojit daný polygon, relativně jednoduchý úkol na první pohled. Z formátu dat OSM popsaného v kapitole 2.1 víme, že:

- V datovém souboru jsou data reprezentována buď jako uzel, cesta nebo relace a měly by v souboru být řazeny v tomto pořadí, ale nemůžeme se na to spolehnout.
- Cesta je tvořena jako seřazený seznam uzlů, s omezením na maximum 2000 uzlů a může tvořit polygon pokud první uzel se rovná poslednímu uzlu v seznamu
- Relace je tvořena seřazeným seznamem tvořeným uzly, cestami a nebo samotnými relacemi (Dokonce může relace obsahovat i sebe samou jako člena). Každý člen tohoto seznamu má popsanou roli, například, že tvoří vnější část polygonu nebo vnitřní v případě hranice státu atd.
- Data pro hranici budou označeny štítky boundary s hodnotou administrative, admin_level s hodnotou 2 a name bude Česká republika

Jako první tedy musíme najít relaci, označenou štítky definující hranici státu. Uzel to být nemůže z toho polygon neuděláme a cesta je pro hranici státu málo, protože má omezení na 2000 uzlů a to na hranici ČR stačit nebude. Procházíme daný soubor element za elementem, projdeme všechny uzly, cesty až dojdeme k relacím a hledáme tu jednu reprezentující hranici státu. Jakmile ji najdeme, tak se podíváme jaké členy má tato relace, vybereme si ty, které tvoří samotný polygon. V relaci jsou členové označeni pouze jako reference, a tedy podle typu člena (uzel, cesta, relace) si uložíme jejich Id a čteme soubor dále a hledáme potřebné členy, dokud nedojdeme do konce souboru. Pokud dojdeme až do konce souboru a stále nám chybí načíst nějaké relace, cesty nebo uzly, musíme číst znova a tento proces opakovat až dokud nebudeme mít všechny potřebné elementy načtené. Jakmile je máme všechny načtené, tak je musíme seřadit podle toho jak byly seřazeny v původní relaci, z relací dostat cesty, z cest uzly a ty pak pospojovat a doufat, že se nám spojí v jeden polygon, který tvoří náš chtěný polygon hranice ČR a ten potom uložit do databáze. Podle toho potom můžeme například určovat, které cesty, body zájmů, adresy atp. patří do ČR, ale to už není součástí tohoto příkladu.

■

Na tomto zdánlivě triviálním příkladu si můžete všimnout složitosti zpracování OSM dat do potřebných formátů (V příkladu to je polygon). Když se tedy vrátím k původní myšlence o zpracování surových datech v paměti a poté uložení do databáze v požadovaném formátu, jsou zde z příkladu vidět zmiňované problémy:

- **Paměťová náročnost** odpovídá složitosti dat, která se pokoušíme načíst, protože například pro zkompletování jedné relace si musíme načíst a někam uložit všechny její členy, než můžeme zpracovat samotnou relaci. To samé platí i v případě cesty, kde si musíme najít a pamatovat všechny uzly, než můžeme pracovat s cestou samotnou. Formát OSM dat v tom nijak nepomůže, ba naopak, ve valné většině případů, pokud potřebujete uzly z cesty nebo členy z relace, už jste v souboru pravděpodobně přeskočili, jako zdánlivě nepotřebná data. A pokud tedy v mém případě, kdy se snažím sestavit cesty, body zájmu a adresy, které podle statistik z taginfo.com tvoří přibližně 65% dat[8], bych v nejhorším možném případě musel mít těch 65% uložených v paměti, než bych s nimi mohl vůbec nějak pracovat a tvořit graf, navíc často ani nemůžu dané elementy z paměti uvolnit, protože nevím zda na ně neodkazují ještě jiné elementy, které jsem ještě nezpracoval. V praxi se ukazovalo, že v paměti je třeba mít uloženo kolem 45%. Navíc relativně hodně paměti je třeba i v případě, že ukládáme pouze id elementu. Například v případě zpracování souboru, pro celou Evropu, jsem si spočítal, že si musím do paměti uložit přes 400 miliónů id uzlů, na které je odkazováno v cestách. Tyto id si potřebuji uložit, abych věděl, které uzly potřebuji načíst ze souboru při dalším čtení souboru. Jednoduchou matematikou, lze tedy spočítat, že pokud bych chtěl mít tyto id v kolekci HashSet<long[9]>, tak bych potřeboval přibližně: $400000000 *$

[8]Toto je hodnota, kterou jsem si spočítal hrubým sečtením všech elementů označených relevantními štítky v celosvětovém měřítku a porovnal s celkovým množstvím elementů v OSM

[9]Datový typ long je 64 bitová reprezentace celého čísla, které může nabývat hodnot -2^63všech až 3^64

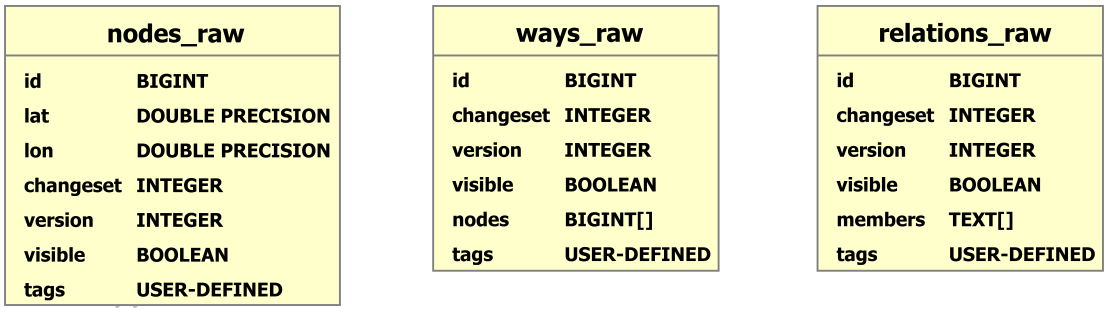
$8/1024/1024 \doteq$ 3GB operační paměti. Ve skutečnosti bude paměti třeba ještě více, protože .NET Framework samotný si přidá nějaký overhead pro správu objektů v paměti a samotná HashSet pokud ji dochází kapacita, tak si nalokuje 2x tolik místa v paměti, než má aktuálně, aby tam mohla umísťovat další prvky a bohužel zrovna třídě HashSet nejde určit v konstruktoru potřebná výchozí kapacita. A uložit id jako int32 (32 bitový integer namísto 64 bitového typu long), taky nelze, protože v OSM je aktuálně nejvyšší id 8E55 ABE9, neboli $2^31 + 240495593$, což je více než vleze do int32, který má nejvyšší hodnotu 2^31.

- **Roztříštěnost** ve formátu OSM způsobuje, již zmíněnou paměťovou náročnost, a taky to, že pokud nečteme pouze elementy typu uzel, tak se nám nepodaří data přečíst v jednom kroku, ale budeme muset číst datový soubor víckrát. V případě cest nejméně dvakrát. V případě relací, se to nedá ani dopředu určit, záleží jak moc velké zanoření najdeme (tedy kolikrát má relace jako člena další relaci), ale v případě, že relace obsahuje pouze cesty, musíme soubor přečíst minimálně 3x (prvně kvůli relace samotné, v druhém čtení načteme cesty, které jsou členy relace a potom ve třetím potřebné uzly tvořící cesty).

Z tohoto důvodu nelze v paměti zpracovávat velké extrakty jako je třeba Evropa. Aby se ovšem takto velké extrakty daly zpracovávat, když už né v paměti, jsem vymyslel dvě řešení. Prvním je cachování surových OSM dat.

Cache surových dat Jako cache surových OSM dat mám na mysli uložiště v paměti, do kterého lze přidávat OSM elementy a pokud se naplní paměť a garbage collector potřebuje nějakou paměť uvolnit, začnou se uvolňovat právě objekty z tohoto uložiště. Cache surových dat se vyplatí hlavně v případě pokud máme k dispozici velké množství operační paměti, kde velkým množství myslím tolik paměti, aby do paměti vlezly nejlépe všechny nebo alespoň většina, takto nacachovaných dat. Cache, kterou jsem vytvořil se skládá ze dvou částí.
První část je generická třída Cache<TKey,TValue>, která slouží jako slovníkové uložiště, neboli ke každému klíčí se dá uložit objekt a pak lze podle klíče tento objekt dostat zpět. Vnitřně tato třída využívá objekty typu WeakRefference, kterému když se předá objekt, který chceme uložit do cache, tak jej garbage collector může, když bude potřebovat uvolnit paměť, smazat. Důležité při použití této, třídy je mít na paměti, že by se neměla používat pro úplně malé třídy, protože pointer na samotný objekt WeakRefference je větší než klasické pointery objektů .NET Frameworku a dále by se tato třída měla používat jenom v případě, že všechny objekty v cache určitě použijeme, ještě jednou, protože i pokud Garbage Collector daný objekt vymaže, stále v cache zůstávají klíče, které se z cache odstraní až tehdy, pokud se pokusíme z cache vytáhnout objekt, který již byl Garbage Collectorem smazán. Do té doby třída Cache neví, zda právě objekt pro tento klíč už Garbace Collector smazal.
A dále pak je důležité mít nějakou zálohu objektu na disku, pokud garbage collector daný objekt smaže, proto jsem vytvořil třídu OsmRawDbCache, která je postavena na zmíněném objektu Cache<TKey,TValue>a při každém vložení objektu do Cache zároveň

asynchronně vloží objekt do databáze a pokud se později pokusíme získat z Cache objekt, který už Garbage Collector smazal, získá jej z databáze a to zcela transparentně aniž, by se nějak změnila práce s Cache. Na obrázku 8 můžete vidět schéma databáze pro uložení nezpracovaných, tzv. surových, OSM dat.



Obrázek 8: UML diagram databáze pro uložení surových OSM dat

V tomto schématu jsem pro uložení štítků použil speciální datový typ hstore[10], který je součástí PostgreSQL databáze od verze 8 a je určen právě pro tento druh dat. U relací jsou členové uloženi jako pole řetězců. Každý člen je určen dvěma řetězci. První řetězec je string, který začíná písmenem N,W nebo R podle toho, jestli je to uzel (Node), cesta (Way) nebo relace (Realation) následovaný svým id. A druhý řetězec je role daného člena v relaci, tedy například pokud by relace měla jako členy uzel s ID = 123456, bez role a relaci s ID = 654321 a rolí ”outer”, byly by v tomto poli přesně 4 řetězce v tomto pořadí a tvaru: ”N123456”, ””, ”R654321”, ”outer”. Tento formát je potom velmi jednoduchý na zpracování a nemusí se tak, vytvářet další tabulka.

Vložit a zpracovat později V některých případech nepomůže ani Cache. V mém případě je to při vytváření zmiňovaného orientovaného grafu pro silniční síť, protože ikdyž jsem schopný si pomocí cache sestavit celou silnici, tak ta je tvořena několika body. Tyto body se spojí a vytvoří se z nich linie, která tvoří geometrii silnice pro vykreslení, ale pro účely routovaích algoritmů je důležitý jenom první a poslední bod dané cesty, protože díky tomu nemusí algoritmus řešit dalších x uzlů z kolika je cesta tvořena, proto se cesta uloží pouze jako hrana, která je dána prvním bodem vytvořené linie a posledním bodem vytvořené linie, tedy jenom dvěma uzly a zbytek se tzv. zminimalizuje do dané geometrie jak můžete vidět na obrázku 9, kde v rámečku vlevo je cesta reprezentována, tak jak je v datech OSM a v rámečku vpravo, jak ji potřebuji reprezentovat pro routovací algoritmy.

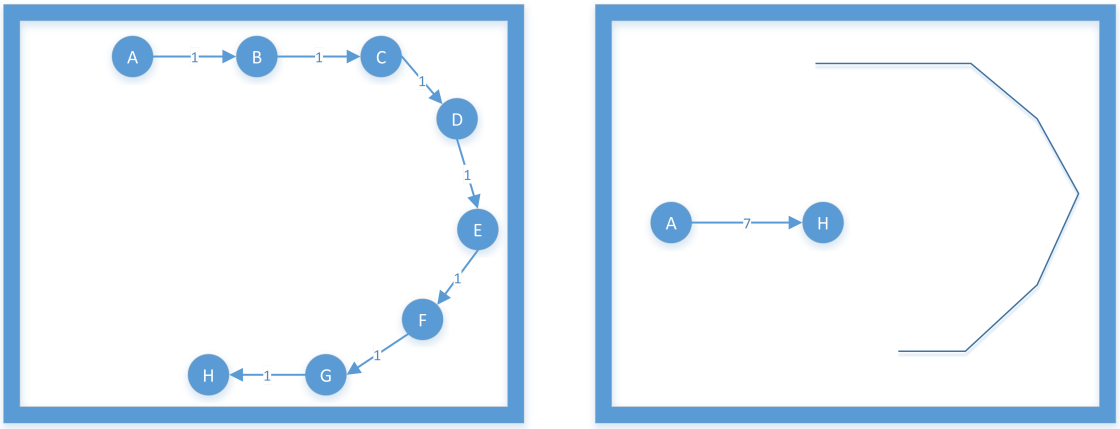
Poznámka 3.3 Čísla u hran v obrázku 9 znamenají vzdálenost mezi uzly, jež hrana spojuje.

Takto vytvořená hrana by se v ideálním případě vložila do databáze společně s dvěma uzly, které ji určují a už by se nemuselo nic víc dělat a máme zpracovanou silnici. Jeno-

[10]Je speciální datový typ, který umožňuje v databázi uložit do jednoho sloupce neomezeně mnoho dvojic typu klíč a hodnota.

může v případě dat z OSM to není konec, protože kvůli tomu jak volně je nastaven způsob ukládání dat v OSM se stane, že přijde další silnice, ktera je opět tvořená několika po sobě jdoucími uzly a zrovna některý uzel mají společný s předchozí již zpracovanou silnicí. Tedy vytvoří v daném uzlu křižovatku, jenomže v tomto okamžiku by to znamenalo velký problém. Problém to je z toho důvodu, že tento uzel, který tvoří křižovatku v databázi neexistuje, protože se zminimalizoval do geometrie a to samé by se s ním stalo s touto silnicí, protože opět by se vzal pouze první a poslední uzel silnice z toho se vytvořila hrana a k té by se uložila její geometrická podoba a tedy tam, kde by jinak měla být křižovatka, by vlastně křižovatka byla jenom vizuálně, pokud by se dané cesty vykreslily, ale v samotném grafu silniční sítě, by se tyto hrany tvářily, že spolu nemají nic společného. Tato situace se nedá ani nijak rozumně vyřešit, muselo by se jedině při vkládání do databáze zjistit zda, už zde neexistuje hrana, jejíž geometrie má společný bod s právě vkládanou silnicí, pokud by se našla, tak geometrii rozdělit, z dané cesty udělat dvě hrany a uložit. To je ovšem velmi náročná operace a celkový import by tak trval velmi dlouho.

Proto jsem udělal kompromis a data se nezpracovávají hned. Tedy jako druhý způsob, jak zpracovat velké OSM data do našeho formátu, bych laicky nazval zpracuj co jde, vlož do databáze a zbytek dopracuj až jsou vloženy všechny potřebné informace. Například v případě zmiňovaných silnic to znamená, že každou silnici vložím do databáze, tak jak mi přijde v OSM datech. Pokud by byla cesta tvořena 3 uzly, řekněme uzly označenými uzly A,B a C, tak danou silnici vložím jako dvě hrany: hranu AB a hranu BC. Jakmile mám takhle vložené všechny silnice, až potom je tzv. zminimalizuji, jak bylo popsáno v příkladu, tedy pokud se v uzlu B nenapojuje jiná silnice, neboli neexistuje zde křižovatka, tak nechám potom databázi tuto hranu zminimalizovat na hranu AC a k ní uložím příslušnou geometrii, kterou bude linka tvořena právě body ABC.



Obrázek 9: Reprezentace cesty pro routovací algoritmy

Pro rychlost tohoto předzpracování je důležité, aby se data do databáze vkládala, co možná nejrychleji, pro tyto účely jsem vytvořil třídu BulkCopy<T>. Třída BulkCopy je třída, která využívá pro vkládání dat SQL příkazu COPY, který je doporučován

přímo autory databáze PostgreSql [10]. Tento příkaz je sice méně flexibilní než příkaz Insert, ale zase na druhou stranu je optimalizován pro vkládání velkého množství dat na jednou, protože nekontroluje každý jednotlivý vložený řádek jak je tomu v případě příkazu insert. Třída BulkCopy potřebuje ke svému fungování celkem 5 věcí. První je ConnectionString na připojení do databáze. Dále je to objekt typu Func<string>pomocí, které se vkládaný objekt serializuje do formátu potřebného pro příkaz COPY. Formátů pro příkaz COPY je několik, nejpoužívanější je Text, Csv a xml. Kde formát Text a CSV se liší jenom jiným oddělovačem dat. Ve své aplikaci pro všechny příkazy COPY používám formát CSV, který je z nabízených nejjednodušší a zabírá nejméně místa. Při serializaci do CSV pro příkaz COPY je nutné, aby jednotlivé hodnoty serializovaného objektu odpovídaly datovým typů sloupců tabulky, do které se vkládá a zároveň je nutné, aby byly serializovány v pořadí, v jakém jsou i sloupce v tabulce. Jako třetí parametr se třídě BulkCopy předává reference na BlockingThreadPool. BlockingThreadPool je třída, kterou jsem napsal speciálně pro účely třídy BulkCopy a je založená na třídách SmartThreadPool od vývojáře Ami Bar z Izraele. BlockingThreadPool má na rozdíl od klasického ThreadPoolu, který je dostupný v .NET Frameworku dvě výhody. Dá se vytvořit jako objekt, tedy není to statická třída a tím se, tak dá vytvořit více ThreadPoolů. A dále jsem mu naimplementoval metodu JoinAll, díky které je možné zablokovat aktuálně vykonávaný kód a počkat tak, až se dokončí veškerá práce, přiřazená konkrétnímu BlockingThreadPoolu. Jelikož BlockingThreadPool používám jenom pro třídy BulkCopy, tedy na vkládání dat do databáze, tak BlockingThreadPool je nastavený tak, že dovolí vytvořit maximálně tolik vláken kolik jader má procesor nebo i více pokud procesor není vytížen na 100%. Tedy počet vláken, které můžou běžet souběžně je omezen na počet jader procesoru a pokud se procesor "nudí"a není vytížen na 100%, tak třída povolí vytvoření i více vláken. Tak lze docílit podle autora třídy SmartThreadPool nejefektivnější využití procesorového času. Dále je nutné třídě BulkCopy specifikovat název tabulky, do které chceme vkládat a poslední parametr, který se třídě nastavuje je počet vkládaných objektů, tedy velikost bufferu pro vkládané objekty. Třída BulkCopy funguje tak, že do ní vkládáme objekty, které chceme vložit do databáze. Jakmile se naplní buffer, tak třída požádá o přidělení vlákna z BlockingThreadPoolu, pokud je jí vlákno přiděleno, tak ho použije a pošle data databáze asynchronně v tomto vláknu, pokud ji vlákno není přiděleno, tak se zařadí do čekací fronty na vlákno a zablokuje se vykonávání kódu do doby, než je vlákno k dispozici.

Srovnání rychlosti Pro ověření, zda je třída BulkCopy opravdu rychlejší jsem provedl měření času pro vložení jednoho miliónu záznamů do databáze. Pro testovací účely jsem použil tabulku relations_raw viz. obrázek 8, do které jsem vkládal 1 milion relací, které jsem si dopředu načetl do paměti z OSM extraktu pro ČR. Pro komunikaci s databází PostgreSQL používám třídy z balíku Npgsql, který je nejpoužívanější .NET data provider pro databázi PostgreSQL. Pro vkládání jsem použil tyto tři způsoby u kterých jsem potom akorát měnil velikost vložených relací najednou, tzv. batchsize:

- Doporučovaný způsob vkládání z dokumentace Npgsql, je pomocí třídy NpgsqlCommand následovně:

```
for (int i = 0; i < 1000000; i++)
{
  NpgsqlCommand command = new NpgsqlCommand("INSERT INTO relations_raw (id,
      changeset, version, visible, members, tags) VALUES (:id, :changeset,
      :version, :visible, :members, :tags)", conn);
  command.Parameters.Add(new NpgsqlParameter("id", NpgsqlDbType.Integer));
  command.Parameters[0].Value = relation[i].Id;
  ...
  command.ExecuteNonQuery();
}
```

Výpis 4: Insert poomcí NpgsqlCommand a kolekce parameters.

Tento způsob má tu nevýhodu, že nejdou jednotlivé inserty spojit do balíku a poslat až je jich nasbíráno více, ale po každém nastavení parametrů je třeba insert odeslat.

- Jako druhý způsob jsem použil opět třídu NpgsqlCommand, akorát tentokrát jsem nepoužil kolekci Parametrs, ale jednotlivé inserty jsem za sebe řetězil ve třídě StringBuilder do formátu "INSERT INTO table (col1,col2) VALUES (1, 'Cheese'),(2, 'Bread'), (3, 'Milk');", dokud jsem neměl požadovanou velikost balíku a při dosažení balíku, jsem takto zřetězené inserty poslal do databáze přes třídu NpgsqlCommand.

```
var sb = new StringBuilder("INSERT INTO relations_raw (id, changeset,
    version, visible, members, tags) VALUES (");
for (int i = 0; i < 1000000; i++)
{
  sb.Append(relation[i].Id);
  sb.Append(", ");
  ...
  if (i % batchSize == 0) {
    command = new NpgsqlCommand(sb.ToString());
    command.ExecuteNonQuery();
    sb.Clear();
    sb.Append("INSERT INTO relations_raw (id, changeset, version, visible,
        members, tags) VALUES (");
  }
}
command = new NpgsqlCommand(sb.ToString());
command.ExecuteNonQuery();
```

Výpis 5: Insert poomcí NpgsqlCommand a řetězením parametrů.

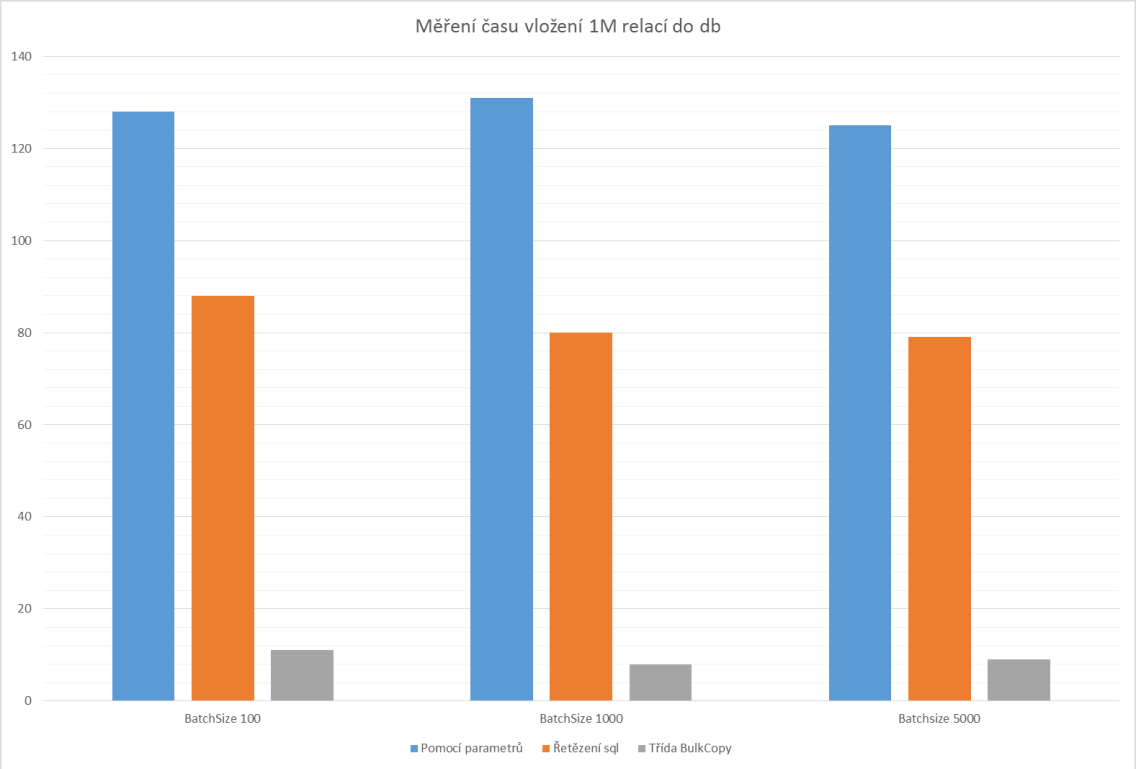
Tento způsob vkládání je navíc nebezpečný, protože se nevalidují vstupy a hrozí, tak SQLInjection útok na databázi, ale pro testovací účely to nevadí.

- A jako poslední vložení pomocí samotné třídy BulkCopy a serializace objektů do CSV, který vypadal takto:

```
var relationsBulk = new BulkCopy<OsmRelation>(ref _dbWorkPool,
    ConnectionString, rel => rel.ToRawCsv(), batchSize, "COPY relations_raw
    FROM STDIN");
for (int i = 0; i < 1000000; i++)
{
  relationsBulk.Insert(relations[i];
}
relationsBulk.Flush();
relationsBulk.Close();
```

Výpis 6: Insert poomcí třídy BulkCopy.

Výsledky měření můžete vidět v grafu na obrázku 10, kde můžete vidět naměřené časy v sekundách. Je jasně vidět, že vkládání pomocí přikazu COPY je mnohem rychlejší, než ostatní možnosti a i přesto, že má své nevýhody, tak pro vkládání velkého množství dat je ideální a je tomu uzpůsoben [10].



Obrázek 10: Výsledky měření času vkládání v sekundách

3.7 Jednotlivé kroky importu a jejich doba trvání

Doposud jsem popsal a odůvodnil jednotlivé části importovacího nástroje. V této sekci popíšu jak do sebe jednotlivé části zapadají, jak importovací nástroj postupuje krok za krokem a jak samotný import trvá dlouho. Časy jednotlivých kroků jsem měřil na svém notebooku, který jsem použil jako testovací sestavu. Notebook má tyto parametry:

- **CPU:** Intel(R) Core(TM) i5 M430 @2,3GHz, 4 jádra
- **RAM:** 3072 MB
- **DISK:** OCZ Vertex3 SSD 256GB

- **OS:** Windows 8 Pro x64
- **SOFTWARE:** PostgreSQL 9.2.4, PostGIS 2.0

3.7.1 Vytvoření polygonů měst

Pokud chceme, aby se při importu adres, kterým chybí město nebo stát a u importu silnic, kterým nešla ze štítků zjistit rychlost, tyto informace doplňovaly na základě polohy vůč musí se před samotným importem nejdříve vytvořit v databázi pomocí třídy PgBoundariesDataTarget v databázi vytvořit polygony měst. Tento krok by mohl být součástí samotného importu, ale vzhledem k tomu, že tato operace nebude probíhat, tak často jako u importu silnic a bodů zájmů (zde se počítá s aktualizací dat každý týden), proto je tento krok zvlášť neboť by zbytečně komplikoval import.

V tomto kroku se čte soubor s OSM daty dokud nejsou, načteny všechny potřebné OSM elementy k sestrojení polygonů do Cache (a zároveň tak i do db viz. sekce 3.6 o cachovaní dat). To vyžaduje většinou celkem 4 čtení souboru s daty. Poté jsou z načtených dat sestrojovány polygony a pomocí třídy BulkCopy vkládány do db. Tento krok na mé testovací sestavě:

- Pro vzorek dat pro celou **Českou republiku** (czech-republic-latest.osm.pbf 319 MB) trval 46 minut 31 sekund a našel a vytvořil celkem 4426 polygonů měst
- Pro vzorek dat pro celou **Evropu** (europe-latest.osm.pbf 10.5 GB) trval 23 hodin 53 minut a vytvořil 95 541 polygonů.

3.7.2 Předzpracování cest, poi a adres a jejich vložení do db

V tomto kroku se čte soubor s OSM daty pouze dvakrát, protože cesty, poi i adresy jsou v OSM datech reprezentovány buď jako uzly nebo cesty, tudíž není více čtení potřeba. Při prvním čtení se čtou pouze elementy typu cesta a pomocí intepreterů se zjišťuje, které cesty se budou importovat a pokud je interpreter vybere, tak se ukládají jejich id i id uzlů, které jsou v cestách odkazovány, do paměti. Při druhém čtení se ze všech uzlů a cest, jejichž id se uložilo během prvního čtení, pomocí interpreterů získají potřebné informace a ty se uloží do databáze. Do databáze se v tomto kroku uloží tyto informace:

- Všechny hrany silnic v nezminimalizované formě s informacemi o počtu jízdních pruhů, třídě silnice, jaká vozidla mohou po silnici jezdit, rychlost pokud se ji podařilo rozpoznat a informace o směru jízdy. Stále zde chybí informace o délce silnice, chybí její minimalizovaná podoba, geometrie a případná maximální rychlost.
- Všechny body zájmů bez jejich lokace a jejich tvaru i s případnou poštovní adresou. Informace o lokaci a jejich tvaru se dá zjistit z tabulky tempnodes, kde jsou uložené

uzly k jednotlivým bodům zájmů, které byly v OSM datech reprezentovány cestou (OSM element way).

- Všechny rozpoznané adresy svázané se zeměpisnou polohou.

Tento krok na mé testovací sestavě:

- pro vzorek dat pro celou **Českou republiku** (czech-republic-latest.osm.pbf 319 MB) trval 23 minut 41 sekund a vytvořil 3 663 155 uzlů, 7 517 301 hran, 1 326 137 bodů zájmů a 2 211 441 poštovních adres.
- pro vzorek dat pro celou **Evropu** (europe-latest.osm.pbf 10.5 GB) trval 14 hodin 1 minutu a vytvořil 64 651 233 uzlů, 135 596 777 hran, 18 159 552 bodů zájmu a 41 milionů adres.

3.7.3 Finalizace dat v databázi

Jakmile jsou všechna potřebná data vložena do databáze, tak probíhá, jejich finalizace, která zahrnuje:

- Odstranění multihran. To jsou hrany, které vzniknou kvůli nekonzistenci dat z OSM, tak že mezi dvěma uzly je dvakrát stejně orientovaná hrana, což fyzicky není možné.
- Minimalizace hran pro routovací algoritmy. (obrázek 9)
- Spočítání délky hran
- Vytvoření geometrií pro zminimalizované cesty.
- Přiřazení lokace a tvaru pro body zájmu, které jsou reprezentovány cestami.

Dále v tomto kroku můžou proběhnout volitelné kroky, které zahrnují dopočítání maximální rychlosti podle polohy cesty vůči polygonům měst a taky zjištění příslušnost adresy k městu a státu pokud tato informace chybí. Při zvolení těchto kroků je nutné, aby v databázi již, vytvořeny tabulky s polygony.

Tento krok na mé testovací sestavě:

- Pro vzorek dat pro celou **Českou republiku** (czech-republic-latest.osm.pbf 319 MB) trval 21 minut a 8 sekund.
- Pro vzorek dat pro celou **Evropu** (europe-latest.osm.pbf 10.5 GB) trval 16 hodin 15 minut.

4 Importovací program

Pro jednoduchost importu jsem vytvořil konzolový program, který umožňuje importy jednoduše provádět. Program se jmenuje Importer.exe a funguje následovně.

Před spuštěním je třeba, aby byl program ve stejné složce, jako jsou ostatní knihovny nutné k fungování programu, nebo cesta k nim byla přidána do systémové proměnné $PATH. Potřebné knihovny jsou tyto: Mapping.Core.dll, Mapping.Data.Core.dll, Mapping.Data.Osm.dll, Mapping.Data.PostGis.dll, C5.dll, log4net.dll, GeoApi.dll, Helpers.dll, ICSharpCode.SharpZipLib.dll, SmartThreadPool.dll, PowerCollections.dll, Plossum CommandLine.dll, Npgsql.dll, NetTopologySuite.dll a Mono.Security.dll. Všechny tyto knihovny jsou součástí balíku s programem.

Program se spouští s následujícími parametry:

zápis parametru	popis
-c, –create	Vytvoří novou databázi pro routing a poi z OSM dat, včetně finalizace
-u, –update	Zaktualizuje existující databázi z OSM dat
-cs, –connectionString	Connection string pro připojení do databáze
-f, –file	Cesta k souboru s OSM daty ve formátu .pbf, .osm nebo .osm.bz2
-h, –help	Vytiskne tuhle tabulku
-p, –polygons	Vytvoří v databázi tabulku s polygony měst a států

Tabulka 6: Tabulka parametrů pro spuštění programu Importer.exe

Pro spuštění je vždy nutné uvést connection string a cestu k souboru, tyto dva parametry mohou být při častém spouštění pro jednoduchost zapsány do konfiguračního souboru Importer.exe.config do a aplikační části konfigurace například takto:

```
<configuration>
  <appSettings>
    <add
      key="connectionString"value="
      Server=127.0.0.1;Userid=postgres;password=****;Database=osm;"
    />
    <add
      key="file"
      value="c:\Users\Martin\Downloads\czech-republic-latest.osm.pbf"
    />
  </appSettings>
</configuration>
```

Výpis 7: Ukázka konfigurace programu

Program upřednostňuje parametry z konzole, tudíž pokud máte v konfiguračním souboru nastavený connection string a zároveň jej uvedete jako parametr při spuštění

programu, tak hodnota connection stringu z konfiguračního souboru bude ignorována. Po nastavení connection stringu a cestě k souboru s daty, je pak nutné uvést jeden z parametrů create, update nebo polygons.

5 Windows Phone aplikace

V této poslední sekci mé diplomové práce představím aplikaci, která využívá právě data zpracované importovacím programem. Nad těmito daty, které se naimportují z OSM je postaveno několik dalších služeb. Ve své aplikaci budu využívat službu pro routování, neboli službu, která umí najít cestu mezi dvěma GPS body a využívá k tomu právě vlastní souborový index vytvořený nad orientovaným grafem silniční sítě. Další aplikací využívaná služba je pro výpočet dojezdové dostupnosti, která opět na grafem silniční sítě, dokáže vypočítat kam se z vybraného místa v jakém čase dá dojet. Potom jsou to služby postavené okolo bodů zájmů a adres. Je to služba na vyhledávání bodů zájmů, podle GPS a fulltextově podle jména a potom na vyhledávání adres. Všechny tyto služby jsou použity a vystaveny ve webové službě na adrese web.floreon.vsb.cz/Routing/Service.asmx. Implementace těchto služeb a webové služby samotné, není součástí této diplomové práce, ale jako taková je součástí připravovaného frameworku. Pro tento framework byla vytvořena v rámci jiné diplomové práce stejně zaměřená aplikace, ale postavená na platformě Android. Obě aplikace slouží jako referenční ukázka použití daných dat na mobilních platformách.

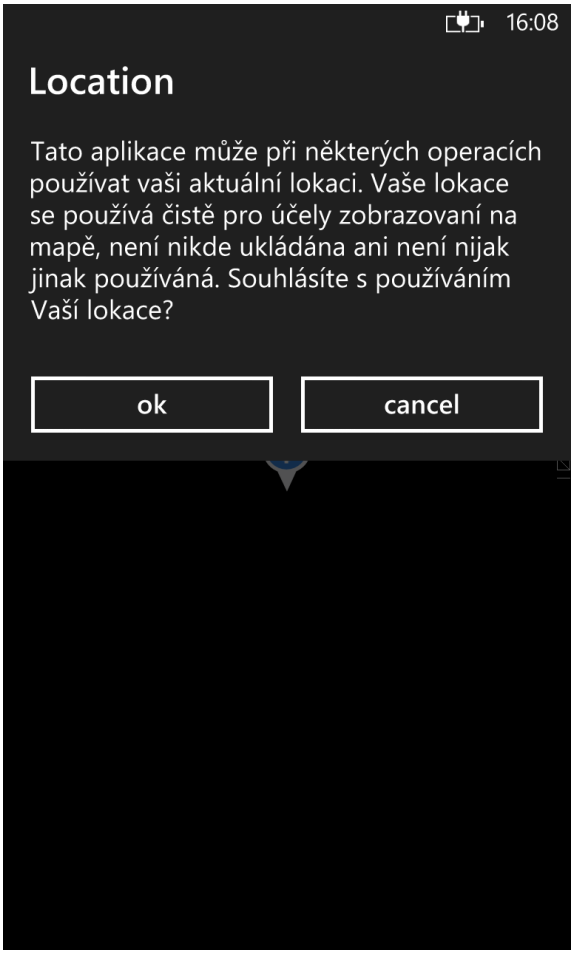
Windows Phone je obchodním názvem operačního systému, který je vyvíjený společností Microsoft a byl vydán v dubnu 2010, tedy je to velmi nová mobilní platforma, která si teprve získává oblibu u svých uživatelů. Windows Phone je nástupce, již nevyvíjeného operačního mobilního systému Windows Mobile, proto může někoho zmást, že první vydaná verze operačního systému Windows Phone je 7. Je to pro to, že Windows Mobile skončil na verzi 6.5 a navíc verzování od verze 7, se shoduje s vydáváním desktopové verze operačního systému Windows, kdy v době vydání prvního Windows Phone, byla desktopová verze Windows rovněž ve verzi 7. Dnes je nejaktuálnější verze operačního systému Windows Phone verze 8, která opět koresponduje s verzováním starého Windows Mobile a taky s verzí desktopové verze Windows, která je rovněž 8. Verze Windows Phone 8 je postavena na stejném jádře jako Windows 8, proto je vyvíjení na obě platformy velmi podobné. Vývojáři aplikací pro Windows Phone mají na výběr ze dvou možností, jak vyvíjet svoji aplikaci. Pro klasické aplikace je k dispozici speciální verze Silverlightu[11] upravená právě pro mobilní platformu Windows Phone. Pro hry je pak k dispozici XNA Framework, který je známý především vývojářům pro herní konzole XBOX od Microsoftu.

Aplikace, kterou jsem naprogramoval je postavena na SDK[12] verze 7.1. Ikdyž nejnovější verze SDK je verze 8, tak verzi 7.1 jsem vybral z důvodů zpětné kompatibility, protože kvůli změně jádra ve verzi Windows Phone 8, nejsou aplikace napsané pro verzi 8 kompatibilní s verzí 7 a nejdou na systému ve verzi 7 spustit. Aplikace je tedy určena pro ty nejstarší telefony s operačním systémem Windows Phone. Tedy všechny telefony

[11] Silverlight je aplikační framework vytvořený pro tvorbu tzv. bohatých internetových aplikací, který je svými funkcemi a zaměřením podobný Adobe Flash frameworku. Grafické rozhraní se definuje pomocí speciálního značkovacího jazyka XAML a samotné jádro aplikací, pak lze vyvíjet v jazyce C# nebo VisualBasic.

[12] SDK je software development kit, neboli vývojářské nástroje pro tvorbu aplikací

s operačním systémem Windows Phone jsou schopné tuto aplikaci spustit, ať už mají v sobě operační systém Windows Phone verze 7 nebo Windows Phone ve verzi 8. V tuto dobu nebyla aplikace ještě publikovaná na Windows Store, odkud by si ji mohl každý stáhnout, prozatím je dostupná pouze na vyžádání pro vybrané uživatele. Aplikace pro své fungování potřebuje aktivní připojení k internetu, právě za účelem využívání zmíněné webové služby. Dále aplikace využívá lokační služby systému Windows Phone. Lokace telefonu se nikde nezaznamenává a už vůbec není spojena s konkrétním telefonem a je využívaná pouze jako parametr při vyhledávání pomocí webové služby, aby se vyhledávání zaměřilo právě na okolí body v okolí telefonu a pak při sledování aktuální polohy pro sledování aktuální pozice na nalezené trase při navigaci. Při prvním spuštění aplikace po nainstalování je nutné s použitím lokačních služeb souhlasit. Žádost o používání lokace telefonu, můžete vidět na obrázku 11. I bez používání lokace telefonu, na rozdíl od internetového připojení, může aplikace omezeně fungovat, tedy i bez souhlasu o používání lokace, se dá aplikace omezeně používat.



Obrázek 11: Žádost o použití lokace telefonu

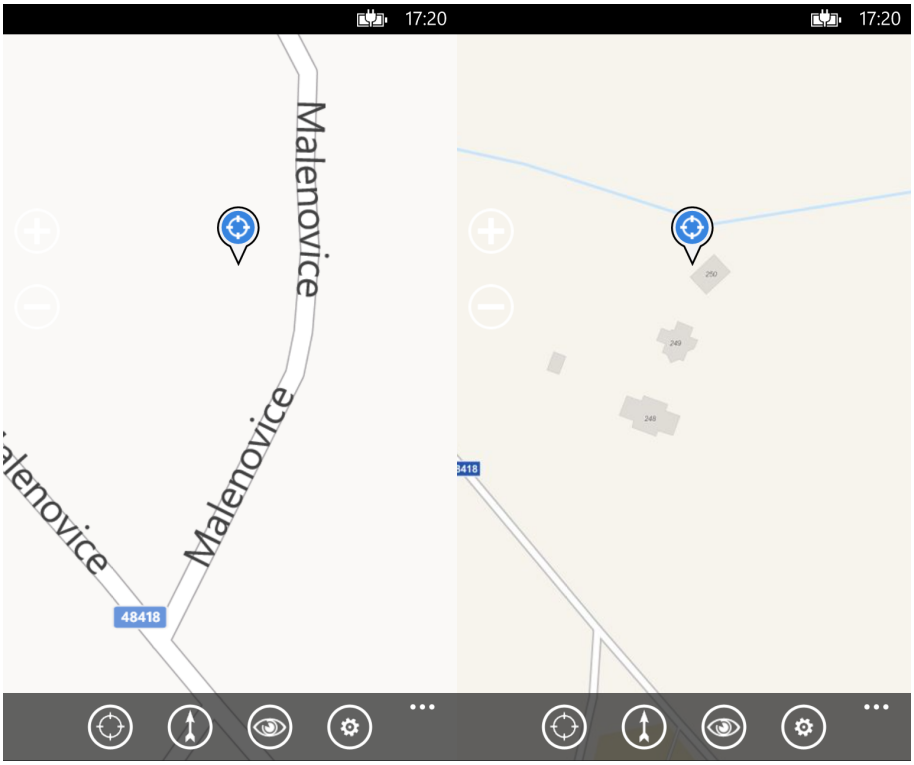
5.1 Použité mapové podklady

V aplikaci jsou na výběr celkem tři druhy mapových podkladů. Jelikož veškerá data jsou založena na datech z projektu OpenStreetMaps, tak jako výchozí mapový podklad jsem zvolil právě podklady generované z dat OSM. Samotný projekt OSM nabízí, již vygenerované podklady ve formátu png. Tyto podklady nejsou, na rozdíl od OSM dat samotných, zdarma k dispozici každému. Veškerá data ze serverů tile.openstreetmap.org jsou dostupné jenom pro testovací účely a nelze je použít v aplikaci bez souhlasu serverových administrátorů (Oficiální znění podmínek používání jsou k dispozici na adrese `http://wiki.openstreetmap.org/wiki/Tile_usage_policy`). Proto jsem se rozhodl použít data z alternativního serveru firmy MapQuest, který dovoluje jejich vygenerované podklady z OSM používat v aplikacích. Tento zdroj je rovněž generován z OSM dat. Získávání podkladů je implementováno ve třídě OsmTileSource, která se stará automaticky o získávání podkladů v potřebném přiblížení a daných souřadnic. Zároveň se tato třída stará o load balancing serverů, dostupných pro tyto OSM podklady.
Mapové podklady generované z OSM dat nemají všude stejnou kvalitu, která se liší podle toho jak dobře je daná oblast v OSM datech zmapována a popsána. V oblastech mimo města se může stát, že nějaká cesta ještě nebyla zmapována nebo přibyl nějaký dům a ten ještě nikdo do OSM dat nepřídal atd. Důkaz můžete vidět například na obrázku 12, kde můžete vidět stejnou lokaci, akorát na levé straně je podklad vygenerovaný projektem Bing Maps ve vektorovém zobrazení silnic, ve kterém je vidět cesta vedoucí kolem aktuální pozice, která v pravé části, kde můžete vidět tutéž lokaci vygenerovanou z OSM dat projektem MapQuest, úplně chybí. Tato lokace je moje bydliště a tato cesta, která v tomto případě vede do centra vesnice a do okolních vesnic, je zde již minimálně 15 let co zde bydlím, pravděpodobně mnohem déle a přesto není zanesena v OSM datech.

Z toho důvodu jsem přidal ještě další dva mapové podklady pro vyšší přesnost. Jeden podklad je Bing Aearial, což jsou satelitní snímky z projektu Bing Maps od společnosti Microsoft a jako třetí je to právě srovnávaný podklad Bing Roads, který zobrazuje pouze silnice. Výběr mapového podkladu a jeho změna je v aplikaci k dispozici na stránce s nastavením viz. obrázek 17, který naleznete v přílohách na konci práce.

5.2 Dostupnost

Jedna z hlavních funkcí aplikace je zobrazování dojezdové dostupnosti. Dojezdovou dostupnost používají hlavně dobrovolné, či profesionální sbory hasičů, záchranáři a nebo třeba policie. V současné verzi pro ukázku jsou v aplikaci dostupné tři režimy dostupnosti. Dojezdová dostupnost dobrovolných hasičů, dojezdová dostupnost profesionálních hasičů a dojezdová dostupnost z jednoho místa daného souřadnicemi GPS. Dále je možnost zvolit nedostupné oblasti, například 100 letá voda, která dostupnost drasticky změní. Dále je možné zvolit maximální rychlost vozidla, podle které se počítají časy dojezdů a v neposlední řadě zda se jedná o nákladní vozidlo a s tím spojené omezení. Pro zapnutí dostupnosti, stačí zajít do nastavení v sekci o dostupnosti vybrat možnost zobrazit na mapě, čímž se otevřou další možnosti nastavení důležité právě pro dostupnost popsané



Obrázek 12: Kvalita podkladů generovaných z OSM(vpravo) v porovnání s Bing Maps(vlevo)

výše. Celou stránku s nastavením si můžete opět prohlédnout na obrázku 18 v přílohách na konci práce.

Veškerý výpočet dostupnosti je prováděn v samotné webové službě, odkud se mimo při startu aplikaci stahují dostupné režimy dostupnosti (dobrovolní hasiči, záchranka atp.) a taky dostupné omezení (100 letá voda, 10 letá voda atp.) pro výpočty dostupnosti. V současné době, aby se ze služby dalo dostupnost zobrazit na mapě, musí provést čtyři kroky, neboli volat službu čtyřikrát:

1. Zavolat na službě metodu WakeUp, tím se služba probudí ze spánku, do kterého se dostane pokud nebyla dlouhou dobu používaná
2. Zjistit dostupné režimy výpočtu dostupnosti
3. Zjistit dostupné omezení pro výpočty dostupnosti
4. Zavolat metodu PrepareAvailability s vybraným nastavení výpočtu dostupnosti

Až po tom co doběhne metoda PrepareAvailability jsou data připravená k zobrazení a pomocí metody GetAvailability, kde parametry jsou opět zoom, a x,y souřadnice v mercatorově zobrazení na mapě, nám služba generuje příslušné výřezy mapy, které

můžeme zobrazovat na mapě. Vygenerované výřezy jsou ve formátu png, tedy formátu, který podporuje průhlednost barev. Čas dojezdu je potom zakódovaný pomocí binární or operace přímo do kódu barvy. Takto vygenerovaný obrázek služba pomocí base64 string kódování potom převede na řetězec a ten vrátí, jako řetězec. Na aplikaci, která potom zobrazuje daný obrázek na mapě, tedy je, ať si převede řetězec zpět na obrázek, projde jej pixel po pixelu, najde vykreslené cesty, z barev pixelu si zjistí čas dojezdu do daného pixelu, podle získaného času pixel přebarví na vhodnou barvu, takhle sestrojí kompletní obrázek a ten zobrazí na mapě. Tento postup pro zobrazení je velmi náročný a jelikož jsem aplikaci testoval i na jednom z prvních telefonů s OS Windows Phone (Samsung Omnia7), tak se ukázalo, že nepřiměřeně zatěžuje telefon zbytečnými výpočty a kazilo to celkový dojem z používání aplikace, protože posouvání v mapě nebylo plynulé, protože při každém posunutí na mapě se muselo takto stáhnout, rozkódovat a zpátky sestrojit několik obrázků. Proto jsem k této službě vytvořil transparentní proxy v podobě ASP.NET HttpHandleru.

5.2.1 HttpHandler pro dostupnost

Handler funguje jako transparentní proxy, to znamená, že klient, když komunikuje s tímto handlerem vůbec nepozná, že tento handler na pozadí komunikuje s webovou službou pro dostupnost. Služba pro dostupnost je založena na kontextu aktuálně komunikujícího uživatele, pro kterého má připravený jiný výstup než pro jiného a k identifikaci využívá soubory Cookies, podle kterých si udržuje danou relaci s klientem. Tento handler převádí soubory cookie ze služby na svou doménu a ty přeposílá klientovi (v tomto případě windows phone aplikaci) a rovněž naopak cookies poslané z klienta, transformuje zase zpátky na cookie pro webovou službu, proto si handler nepotřebuje udržovat žádné data k udržení kontextu, jelikož ten je udržován na službě samotné. Tento handler ulehčuje klientům od těchto operací:

- Sám službu vzbudí, pokud služba spí, voláním metody WakeUp na službě
- Není třeba před získáváním výřezu mapy nejdříve volat metodu PrepareAvailability pro přípravu dat. Handler sám pozná ze souborů cookies, že data ještě nejsou na službě připravena a sám tuto metodu zavolá s potřebnými parametry. Rovněž pozná, že nastavení se od předchozího volání změnilo a tudíž je třeba zavolat PrepareAvailability znovu s novými parametry.
- Handler sám zpracuje vrácený výřez mapy a překreslí vrácený obrázek a vrátí jej hotový přímo klientovi a tomu, tak zbývá ho jen zobrazit na mapě, nejsou třeba, žádné další operace s tímto obrázkem.

Aktuálně handler běží v cloudu Azure od Microsoftu na adrese `http://floreon.azurewebsites.net/AvailabilityTiles.ashx`, tento handler má velmi malé požadavky na výpočetní výkon a nulové požadavky na uložiště, proto může běžet v bezplatné verzi hostování v cloudu Azure, jediné, co se může dostat mimo bezplatnou verzi je množství poslaných dat, ale to zatím není problém. Velikost jedné HTTP odpovědi, pro získání

výřezu pro zobrazení do mapy při použité velikost 256x256 pixelů má dle mých měření ve Wiresharku[13] průměrně 28 kB, pokud zařízení nepodporuje gzip encoding a v případě použití gzip komprese mají odpovědi v průměru 13 kB. Samozřejmě tato naměřená hodnota se může od průměru lišit podle zrovna zobrazované části, ale v průmětu je to tato velikost.

Parametry používané při volání handleru můžete vidět v tabulce 7. Url pro volání handleru může vypadat například takto: `http://floreon.azurewebsites.net/AvailabilityTiles.ashx?startgroup=-8588443167173212406&startgroup=-8588443166286431121&maxspeed=200&tir=True&zoom=13&size=256&x=4423&y=2778`

název parametru	popis
startgroup	těchto parametrů může být nastaveno více najednou a specifikují id skupin pro výpočet dostupnosti (hasiči, záchranka atd.), získané z volání metody služby GetStartLocationGroups
lat a lon	specifikace zeměpisné šířky a délky v E6 formátu, pokud je výpočet dostupnosti místo skupin z jednoho místa
restriction	id omezení pro výpočet dostupnosti získané z volání metody GetRestrictions
maxspeed	rychlost vozidla v kilometrech za hodinu
tir	boolean specifikující zda se má počítat dostupnost pro nákladní vozidlo, či ne
zoom	hodnota přiblížení v mercatorově zobrazení
size	požadovaná velikost získaného výřezu v pixelech (výška i šířka). Možné hodnoty 128,256,512,1024
x	x-ová souřadnice v mercatorově zobrazení
y	y-ová souřadnice v mercatorově zobrazení

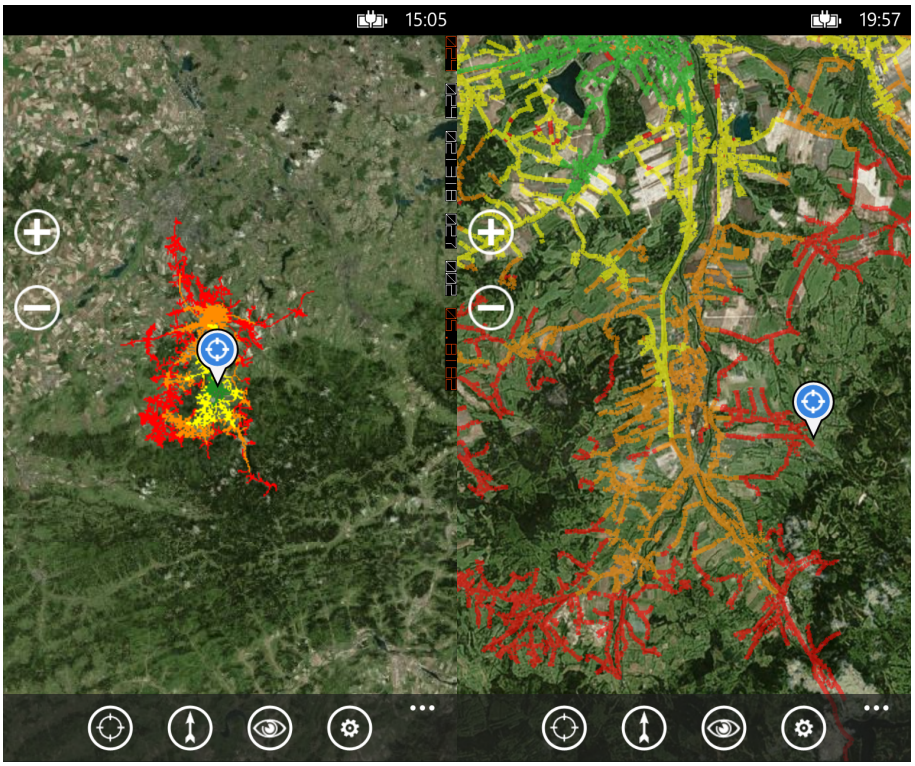
Tabulka 7: Tabulka parametrů pro volání handleru pro mapové výřezy

Handler má explicitně nastaveno, že všechny cesty kde je dojezd do 5 minut jsou vybarveny zeleně, mezi 5 a 10 minutami žlutě, mezi 10 a 15 minutami oranžově a nad 15 minut červeně. Konkrétní výstup handleru můžete vidět na obrázku 13 a zobrazení v aplikaci na mapě na obrázku 14. Díky handleru je práce s mapou se zobrazenou dostpuností plynulá a má rychlou odezvu. Jediná delší odezva při změně nastavení, nebo když se dostupnost zapne poprvé, protože se musí čekat až webová služba dostupnost přepočítá s novým nastavením, to může trvat až 15 sekund, ale ve většině případů je to spočítáno do 3 sekund. Cachování obrázků si dělá mapová komponenta (Map Control) sama. Při spuštění aplikace se stahuje celkem 35 výřezů. Přesné fungování cache výřezů u této komponenty není přesně popsáno, protože se dle slov Kristoffer Henriksson

[13]Wireshark je nástroj pro analýzu síťového provozu, ke stažení na www.wireshark.org.

z vývojářské divize Microsoftu [12] bude měnit. Zatím dle mých odhadů z debugeru, podle počtu volání HttpHandleru, je cache nastavena stejně, jako je nastavena cache pro Internet Explorer v systému. Životnost výřezů je dána časově a je ukládána do dočasných souborů ve speciálním zkomprimovaném formátu jako v případě cache pro stránky v Internet Exploreru.

Obrázek 13: Výstup handleru pro výřezy map u dojezdové dostupnosti



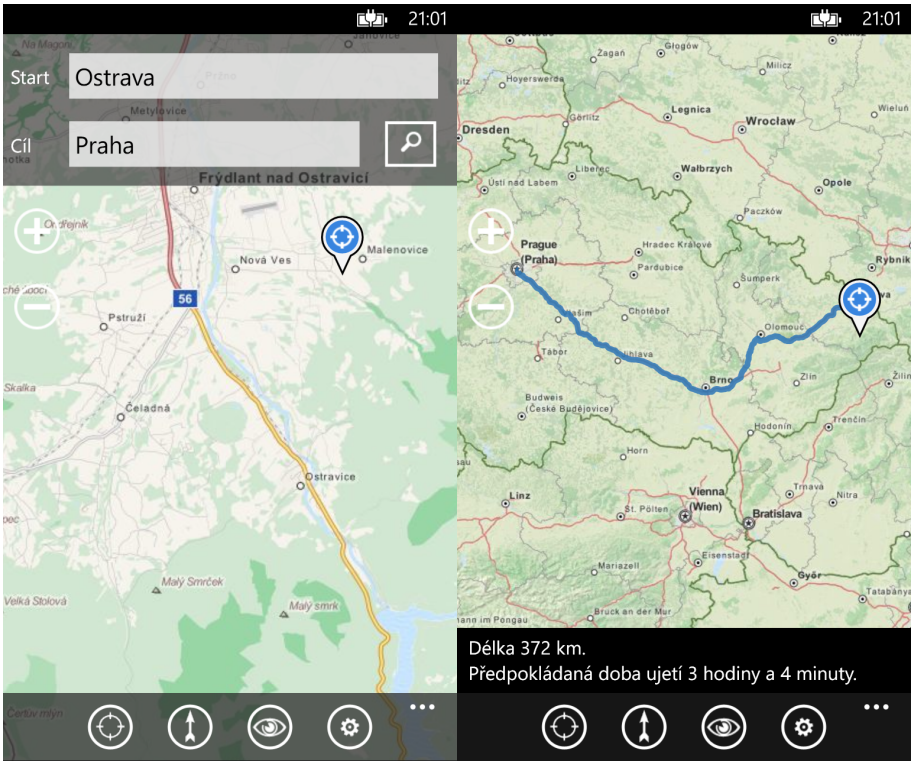
Obrázek 14: Zobrazené výřezy na mapě v mobilní aplikaci. Vlevo z jednoho místa. Vpravo dojezd dobrovolných hasičů. (Horská oblast pod Lysou horou)

5.3 Hledání trasy a navigace

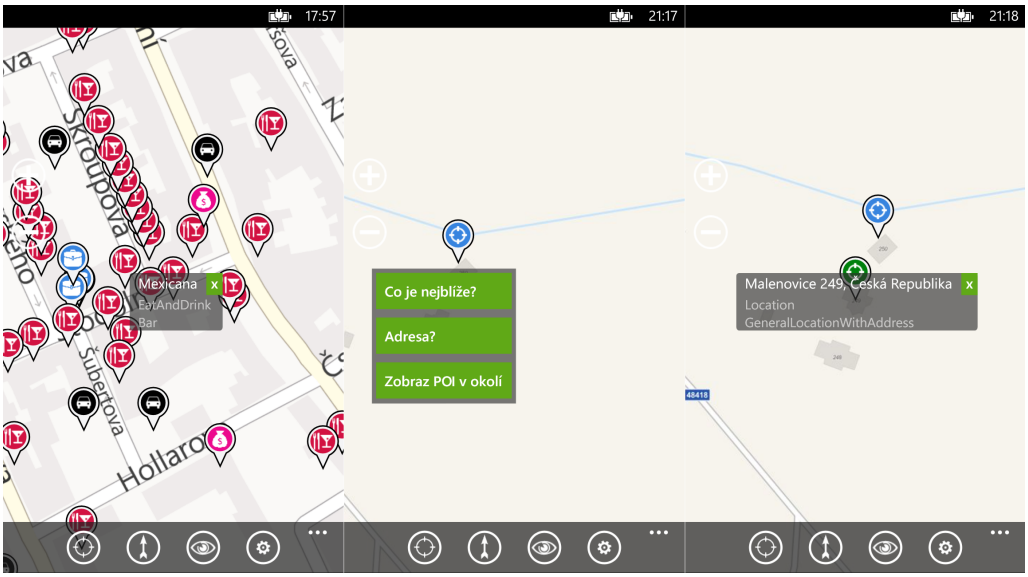
V aplikaci je možné vyhledávání trasy a poté sledovat pohyb po nalezené trase. Vyhledávání trasy v aplikaci funguje po stisku tlačítka najdi trasu na hlavní obrazovce, které má tvar šipky. Po stisku se objeví dvě textové pole, kam se zadává start a cíl. Start a cíl se může zadat buď jako GPS souřadnice, nebo se může zadat jako název místa. Pokud zadáte start nebo cíl jako název místa, telefon se pokusí nejdříve za pomoci webové služby přeložit tento název a získat k němu korespondující souřadnice. Telefon zkouší nejprve místo interpretovat jako adresu a tu vyhledávat pomocí webové služby, pokud se nepovede adresu najít, tak se pokusí pomocí služby vyhledávat v bodech zájmů, pokud se ani to nepovede, tak se telefon pokusí daný název přeložit pomoci vyhledávače Bing od Microsoftu v tomto pořadí. Pokud ani jedna služba nedovede místo přeložit na souřadnice GPS, budete vyzváni o zadání místa znovu, dokud se nepodaří obě dvě textová pole interpretovat na souřadnice GPS. Jakmile se toto povede, GPS souřadnice jsou poslány do naši webové služby a ta najde danou trasu a vrátí nám ji jako výsledek s doplňujícími informacemi o délce a času ujetí. Takto vrácena trasa je potom zobrazena na mapě i s doplňujícími informacemi. Momentálně služba neumí vytvořit tzv. turn-by-turn instrukce, neboli jednotlivé instrukce typu, za 300m odbočte vlevo, jeďte rovně 1km atp. Proto pokud chcete alespoň orientačně sledovat, jestli jedete správně, můžete v aplikačním menu zapnout sledování polohy a sledovat zda se pohybujete po vyhledané trase. Vyhledání trasy můžete vidět na obrázku 15.

5.4 Body zájmu a adresy

Aplikace umí vyhledávat i v bodech zájmů a adresách. Toto představuje jednoduché volání webové služby a zobrazení výsledku na mapě. Pro vyhledávání adresy daného místa, stačí na daném místě podržet prst a počkat až se objeví kontextové menu. Z kontextového menu potom vyberte možnost adresa, tímto se aplikace doptá služby na nejbližší známou adresu k bodu kde jste drželi prst. Stejný postup platí při hledání nejbližšího bodu zájmu nebo pro zobrazení nejbližších 100 bodů zájmů. Pokud chcete vyhledávat specifické body zájmu použijte aplikační menu a položku hledat body zájmů, kde můžete vaše hledání specifikovat na jednotlivé kategorie bodů zájmů. Jak vyhledávání vypadá v praxi můžete vidět na obrázku 16. Po kliknutí na jednotlivé nalezené body, se můžete dozvědět více informací o daném bodu.



Obrázek 15: Vyhledání trasy v aplikaci



Obrázek 16: Vyhledání budů zájmu a adres

6 Závěr

V této práci jsem vytvořil nástroj na práci s daty z projektu OpenStreetMap, který se zaměřuje hlavně na data týkající se navigace, bodů zájmů a poštovních adres. Při implementaci tohoto nástroje jsem odhalil spoustu problémů, které OSM data mají, jako je jejich nekonzistence, roztříštěnost, složitost zpracování a samotná kvalita. Z této práce s těmito daty vyplývá, že myšlenka a směr, kam projekt OSM směřuje je dobrý, ale, že kvalitou nemusí pro specifické projekty ještě stačit a vyplatí se spíše šáhnout po placených variantách nebo si ještě počkat až se data v OSM naplní na požadovanou kvalitu. Tato práce mimo jiné ukazuje, že tyto data i přes všechny své nešvary se již dnes dají použít a vytvořená aplikace na Windows Phone, která s těmito daty pracuje je toho praktickým důkazem.

Martin Škuta

7 Reference

[1] Lorenzo Alberton. Graphs in the database: Sql meets social networks. online, Září 2009. URL: http://techportal.inviqa.com/2009/09/07/graphs-in-the-database-sql-meets-social-networks/.

[2] Mohammad Beydoun and RamziA. Haraty. Directed graph representation and traversal in relational databases. In Filip Zavoral, Jakub Yaghob, Pit Pichappan, and Eyas El-Qawasmeh, editors, *Networked Digital Technologies*, volume 88 of *Communications in Computer and Information Science*, pages 443–455. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-14306-9_44.

[3] Judith Bishopová. *C# - návrhové vzory*. Zoner Press, 2010.

[4] Gordon Carrie. Basic geographic calculations and comparisons. online, Červen 2009. URL: http://www.tkgeomap.org/man/geography.3.html.

[5] Steve Coast. Yahoo! aerial imagery in osm. online, Prosinec 2006. URL: http://blog.openstreetmap.org/2006/12/04/yahoo-aerial-imagery-in-osm/.

[6] Steve Coast. And donate entire netherlands to openstreetmap. online, Červen 2007. URL: http://blog.openstreetmap.org/2007/07/04/and-donate-entire-netherlands-to-openstreetmap/.

[7] Daniel Cooper. Craigslist quietly switching to openstreetmap data. online, Srpen 2012. URL: http://www.engadget.com/2012/08/28/craigslist-open-street-map/.

[8] Mike Fossum. Websites bypassing google maps due to fees. online, Březen 2012. URL: http://www.webpronews.com/websites-bypassing-google-maps-due-to-new-fees-2012-03.

[9] I.A. Getting. Perspective/navigation-the global positioning system. *Spectrum, IEEE*, 30(12):36–38, 1993. doi:10.1109/6.272176.

[10] The PostgreSQL Global Development Group. Populating a database. online, Březen 2013. URL: http://www.postgresql.org/docs/current/interactive/populate.html.

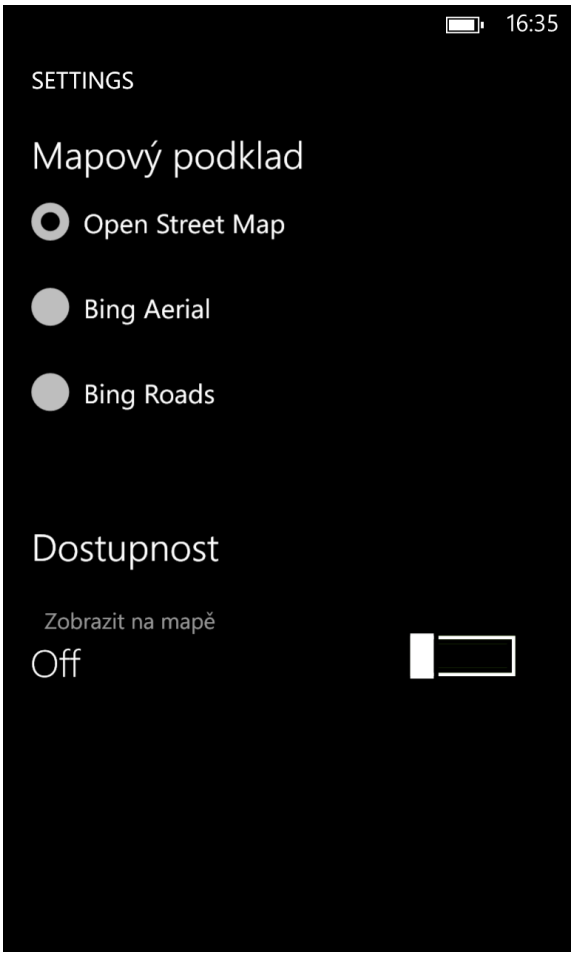
[11] M. Hazas, J. Scott, and J. Krumm. Location-aware computing comes of age. *Computer*, 37(2):95–97, 2004. doi:10.1109/MC.2004.1266301.

[12] Kristoffer Henriksson. Maptilelayer caching. online, Srpen 2012. URL: http://social.msdn.microsoft.com/Forums/en-US/5f3d15ba-3b43-4403-a3de-cb4ea70af8c4/maptilelayer-caching.

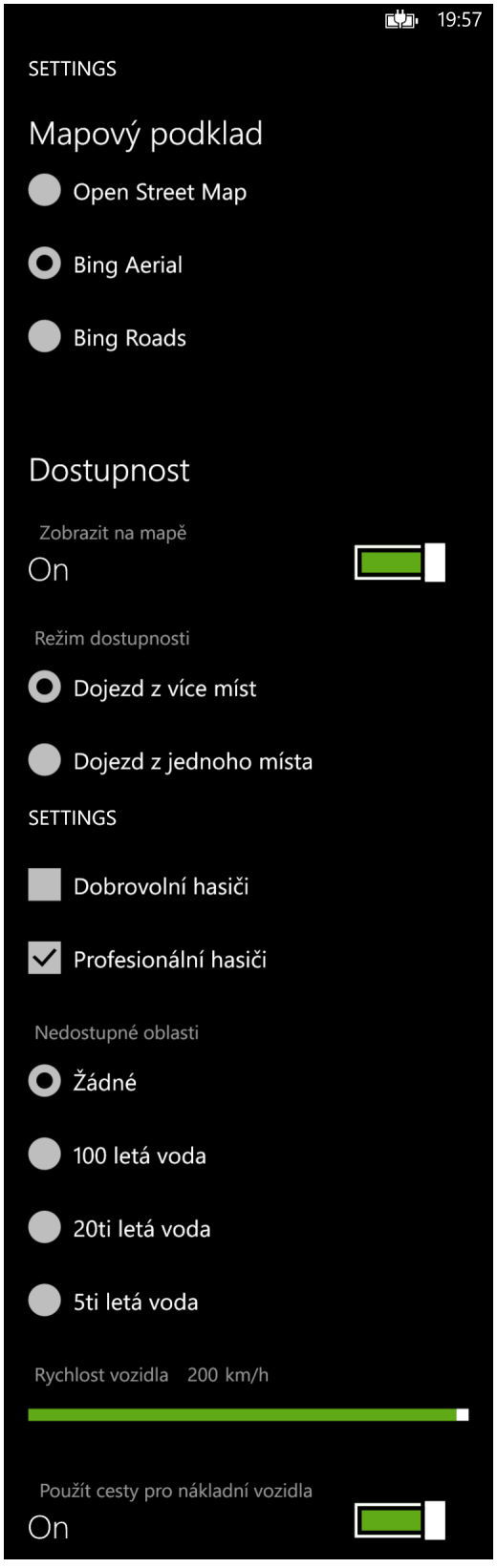
[13] Nathan Ingraham. Apple using tomtom and openstreetmap data in ios 6 maps app. online, Červen 2012. URL: http://www.theverge.com/2012/6/11/3078987/apple-tomtom-openstreemap-ios-6-maps-app.

[14] Jiří Sedláček. *Úvod do teorie grafů*. Academia, 1977.

A Další screenshoty z aplikace pro Windows Phone



Obrázek 17: Stránka s nastavením pro výběr mapových podkladů



Obrázek 18: Nastavení dostupnosti

Obrázek 19: Loadin screen aplikace pro windows phone